Teljesértékű tankönyv és forrásmű

minden komoly és leendő Delphi programozónak

Marco Cantù

Delphi 3

Mesteri Szinten

II. KÖTET

SYBEX®

KISKAPU

Kiskapu Kiadó

KISKAPU

Hazai és angol nyelvű szakkönyvek kis- és nagykereskedése

Szeretettel várjuk önöket hazai és külföldi szakkönyv-kiadók kínálatával, hazai és angol nyelvű számítástechnikai magazinokkal.

Társaságunk a következő kiadók termékeit forgalmazza:

Addison Wesley Adobe Press Angster Apollon AurumDTP Bagolyvár Borland Press Hayden Books HungaLibri IDG Books Julianus Kalibán Kiskapu Kossuth Lobenwein Lövey LSI M&T Books Macmillan Macromedia Manning Microsoft Press MIS:Press Műszaki Múzsák NeTeN New Rider's Novotrade O'Reilly Openinfo Oscar Panem Park PC Start Prentice Hall Proxerv PTR PWS QUE Sams SamsNet Sigs Books Sors Bona Springer Sybex Szak Tertia Tesloff Babilon Thompson Ventana Virgil Waite Group Wiley's Ziff-Davis Press

Ezen kívül igény szerint más szakkönyveket is beszerzünk

Kedves Olvasó!

A könyv anyagának zárása után kaptunk két fontos információt, melyet szeretnénk megosztani Önökkel.

A Magyar Delphi Levelezési Lista új címre költözik át. Az előszóban is említett

```
http://www.szechenyi-nkzsa.sulinet.hu/delphi
```

címen találhatnak bővebb információt a költözéssel kapcsolatban.

A szerző honlapjának címe is megváltozott, az új cím:

```
http://www.marcocantu.com/
```

Üdvözlettel,

A kiadó

Delphi 3

Mesteri Szinten

II. KÖTET

Kiskapu Kiadó

A kiadvány a következő angol eredeti alapján készült:
Marco Cantù: Mastering Delphi 3 second edition

Copyright © 1997 Sybex Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

Associate Publisher: *Gary Masters*
Acquisitions Manager: *Krisine Plachy*
Desktop Publisher: *Franz Baumhackl*
Technical Editor: *Juancarlo Añez*
Editor: *James A. Compton*
Graphic Illustrator: *Inbar Berman*
Production Coordinator: *Alexa Riggs*
Indexer: *Nancy Guenther*
Proofreaders: *Jennifer Matzger, Theresa Gonzalez, Michael Tom*
Acquisitions & Developmental Editor: *Peter Kuhns*

SYBEX is a registered trademark of SYBEX Inc.
Mastering is a trademark of SYBEX Inc.

Microsoft Windows, Windows 95 and Windows NT are registered trademarks of Microsoft Corporation.
Netscape Communications, the Netscape Communications logo, Netscape, and Netscape Navigator are trademarks of Netscape Communications Corporation.

A szerző és a kiadó a lehető legnagyobb körültekintéssel járt el e kiadvány elkészítésekor, a könyv tartalmát a szoftver végső kiadásához igazították, ahol ez lehetséges volt. A kézirat bizonyos részei a szóban forgó szoftverek úgynevezett „prerelease" verzióin alapul, mely programokat azok gyártói biztosítottak a kiadvány elkészítéséhez. Sem a szerző, sem a kiadó nem vállal semminemű felelősséget vagy garanciát a könyv tartalmával, teljességével kapcsolatban. Sem a szerző, sem a kiadó nem vonható felelősségre bármilyen baleset vagy káresemény miatt, mely közvetve vagy közvetlenül kapcsolatba hozható e kiadvánnyal.

Magyar változat © 1998 Kiskapu Kft. Minden jog fenntartva!
Hungarian edition © 1998 Kiskapu Kft. All rights reserved!

Sorozatszerkesztő: *Szy György*
Műszaki szerkesztő: *Troják Miklós*
Fordításvezető: *Tanács Lajos*
Lektorok: *Troják Miklós, Szy György*
Fordítók: *Illés Róbert, Kisteleki Róbert, Ludányi Levente, Mihalicza József, Nagy Tamás, Tanács Lajos*

Felelős kiadó a Kiskapu Kft. ügyvezető igazgatója
© 1998 Kiskapu Kft.
1081 Budapest Népszínház u. 29.
Tel: (+36-1) 303-9119 Fax: (+36-1) 303-1619
http://www.kiskapu.hu/ e-mail: kiskapu@kiskapu.hu

ISBN: 963 03 5349 0
Összesített ISBN: 963 03 5347 4 ö

Előszó a Második Kötethez

Ahogy ez lenni szokott, az egyik szemem nevet, a másik sír. Örülök, mert végre sikerült megjelentetni ezt a remek könyvet kis hazánkban, ugyanakkor szomorú vagyok, hisz ismét egy verzióval lemaradtunk külföldtől. Mire mi megjelentettük a hármas verziót, a Delphi 4 már elindul világhódító útjára.

De félre bánat! Mindenképpen örülök, hiszen ez a könyv a programnyelvekkel foglalkozó sorozat első tagja. Remélem, hogy olvasóink megkedvelik a sorozatot, és megírják, elmondják véleményüket, tapaszatalatiakat.

Szeretnénk mi is a tőlünk tellő legtöbb segítséget megadni mindenkinek e programnyelv elsajátításához. Ezért itt közlök egy-két fontosabb címet, mely esetleg segítséget jelenthet a programozóknak válaszaik keresése közben:

Magyarországon működik egy Delphi levelezési lista. E listával kapcsolatban további információkat olvashatnak az alábbi Internet címen:

```
http://www.szechenyi-nkzsa.sulinet.hu/delphi
```

A levelezési listára az alábbi e-mail címre küldött, a tárgy (subject) rovatba írt subscribe szöveggel lehet:

```
delphi@sop3.sop.matav.hu
```

Ezeket, és további információkat találhatnak meg az angol nyelvű könyv hivatalos honlapján:

```
http://ourworld.compuserve.com/homepages/marcocantu/
http://www.sybex.com/
```

Valamint a hazai változat hivatalos honlapján:

```
http://www.kiskapu.hu/konyvek/delphi/
```

Amennyiben a szerzőnek kívánnak írni levelet (angol nyelven):

```
marcocantu@compuserve.com
```

A Kiskapu Kiadó is örömmel várja az Olvasói észrevételeket:

```
kiskapu@kiskapu.hu
```

Jó olvasást és jó programozást kívánva,

A sorozatszerkesztő,

Szy György

Tartalomjegyzék

KOMPONENSEK ÉS KÖNYVTÁRAK 1
Komponensek létrehozása 5
A VCL kibővítése 5
A komponensektől a csomagokig 6
A komponensírás szabályai 8
Az első komponens elkészítése 9
A Font kombinált lista 10
Csomagok létrehozása 13
Hogyan épül fel egy csomag? 14
A fejezet komponenseinek telepítése 16
A Font kombinált lista használata 16
Tabulált listapanel készítése 18
A készített listapanel komponens kipróbálása 19
Új komponensek készítése 21
Egy grafikus komponens építése lépésről-lépésre . . 21
Felsorolás típusú tulajdonság definiálása 22
A Paint metódus 25
A tesztprogram 26
Osztály típusú tulajdonságok hozzáadása a komponenshez . . 28
A tesztprogram átírása 32
Saját esemény definiálása 33
Az OnArrowDblClick esemény kipróbálása 35
Bitkép hozzáadása a komponenspalettához 37
Az Arrow komponens utolsó próbája 38
A hibák kijavítása az Arrow komponensben 39
Nyilak és alakzatok 40
Az óra komponens 42
Tömbtulajdonság definiálása 44
Az új tabulált lista 44
A párbeszédablak mértékegységeinek használata 48
Fejléc és a tabulált listapanel 49
Egy nem vizuális komponens készítése 51
Párbeszédablak a komponensben 52
Nem vizuális komponensek használata 55
Hogyan tovább? 56

A komponensek és a ToolsAPI 61
Tulajdonság-szerkesztők (property editor) készítése .61
A tulajdonság-szerkesztők típusai 61
Szerkesztő a TabsString tulajdonsághoz 63
A tulajdonság-szerkesztő telepítése 65
Komponens-szerkesztő készítése 67
A TComponentEditor osztály és leszármazottai 68
Komponens-szerkesztő a listához 68
A komponens-szerkesztő regisztrálása 71
Egyszerű varázslók készítése 71
A TIExpert osztály és leszármazottai 72
Egy egyszerű varázsló 73
A ToolsAPI további lehetőségei 75
Tulajdonságok elérése név szerint 76
Hogyan tovább? 79

Dinamikusan szerkesztett könyvtárak 83
A DLL-ek szerepe Windows alatt 83
Mi is az a dinamikus szerkesztés? 83
Mire való a DLL? 85
A rendszer DLL-jei 87
Csomagok (package-ek) statikus és dinamikus használata .88
Különbségek a DLL-ek és az EXE-k között 88
A DLL készítésének alapszabályai 89
A Win16 és a Win32 DLL-ek 89
A meglévő DLL-ek használata 90
DLL készítése C++-ban 91
C++ DLL függvény deklarálása Delphiben 92
DLL létrehozása Delphiben 95
Az első egyszerű Delphi DLL 96
Delphi DLL hívása 97
Delphi form a DLL-ben 99
A DLL meghívása Delphiből 101
Delphi DLL hívása a Visual Basic for Applications nyelvből 102
Ikonokat tartalmazó DLL készítése 103
Ikonok betöltése DLL-ből 104
DLL függvény meghívása futásidőben 106
A DLL a memóriában 108
Hogyan tovább? 110

OLE és COM 113
Mi is voltaképpen az OLE és a COM? 113
Objektumok a DLL-ekben 115
Osztályok egy DLL-en belül 115
Osztály használata DLL-ből 117
Az IUnknown implementálása 120
IUnknown metódusok implementálása 120
Globális egyedi azonosítók 122
Az osztálygyárak szerepe 125
Delphi 3 COM interfészek használata 126
A TComObject osztály 126
COM objektumok inicializálása 128
A COM kliens program frissítése 131
Az interfész tulajdonságok használata 134
Shell interfész használata 135
Shell csatolások (vagy parancsikonok) készítése 135
Megerősítést kérő ablak Pascal fájlokhoz 137
Hogyan tovább? 142

Az OLE Automation és OLE Document technikák 145
Az OLE Automation 145
Adatok küldése a Wordnek 146
Adat küldése a Microsoft Wordnek egy adatbázisból 147
OLE Automation szerver írása 150
A típuskönyvtárak 150
A típuskönyvtár-szerkesztő 152
Az Automation szerver regisztrálása 156
Írjunk klienst a szerverünkhöz! 156
Az OLE dokumentum 159
Az OleContainer komponens használata 161
Egy egyszerű OLE tároló 161
OLE tároló bővítése menüsorral 164
A vizuális szerkesztés és az eszköztárak 166
A szabványos OLE párbeszédablakok 167
Objektumok betöltése fájlból és mentése fájlba 169
Többszörös OLE tárolók 170
Hogyan tovább? 172

ActiveX kontrolok használata és készítése 175
ActiveX vezérlők bemutatása 175
ActiveX vezérlők kontra Delphi komponensek 177
ActiveX vezérlők használata Delphiben 177

ActiveX vezérlők telepítése *178*
A TOleControl osztály *180*
ActiveX vezérlők használata *180*
Grafikonok készítése *182*
ActiveX vezérlő írása 183
Az Arrow komponens átírása ActiveX vezérlővé *183*
Új tulajdonságok hozzáadása egy vezérlőhöz *188*
Tulajdonságlap hozzáadása a komponenshez *190*
Hogyan tovább? 194

Internetes programozás 197
HTML fájlok böngészése 197
HTML fájlok készítése 202
Egy egyszerű HTML oldal generálása *203*
HTML táblák készítése *206*
HTML oldalak kívánság szerinti készítése *210*
ActiveFormok létrehozása 211
ActiveX form szerepe egy weblapon *215*
Többlapos ActiveFormok *216*
Hogyan tovább? 217

FEJLETT DELPHI PROGRAMOZÁS 219
Az alkalmazások szerkezete 223
Az Application objektum használata 223
Az alkalmazás-ablak megjelenítése *226*
Az alkalmazások rendszermenüje *227*
Alkalmazások és formok aktiválása *229*
Egy alkalmazás példányainak kezelése 230
A fő ablak egy másolatának felkutatása *231*
Az előző főform aktiválása *231*
A felhasználó által definiált Windows üzenetek kezelése *233*
Keresés az ablaklistában *234*
Kölcsönös kizárás (mutex) használata *236*
Komponensek nélküli programozás 236
A legkisebb Delphi program? *237*
A parancssor olvasása *238*
Események, üzenetek, és párhuzamos programozás a Windowsban 240
Eseménykezelt programozás *240*
A Windows üzenetkézbesítése *241*
Egy óra elkészítése időzítő segítségével 242

Egy grafikus óra 244
A másodpercmutató megrajzolása rasztermíveletekkel 247
A Windows rasztermódjai 247
A grafikus óra új változata 248
Háttérszámítás és párhuzamos programozás 249
Háttérszámítás 250
Prímszámok keresése 250
A „homokórás" megközelítés 251
Háttértevékenység időzítőkkel 253
Üzenetek feldolgozása a háttérben 254
Háttérszámolás 255
Párhuzamos programozás a Delphiben 256
A TThread osztály 257
Az első példa 258
Egy zároló példa 259
Szinkronizálási lehetőségek 261
A szálak prioritása 261
Szálak szinkronizálása 263
Várakozás egy szálra 264
Windows összehangolási technikák 268
Egy példa elkészítése 269
Az egyszerű szál 270
Kritikus szakaszok használata 271
Mutex használata 272
A TCriticalSection VCL objektum használata 272
A Screen objektum használata 273
Képernyőinformációk lekérdezése 273
A formlista kezelése 274
Állapotinformációk elmentése 277
Windows INI fájlok használata 278
A Registry használata 281
Hogyan tovább? 287

Egy program működésének vizsgálata 291
A hibakereső (Debugger) használata 291
Hibakeresési információ 292
A töréspontok beállítása 293
Az értékek vizsgálata 299
Bővebben a töréspontokról 301
Nyomkövetés a VCL forráskódban 303
Hibakeresés, másképpen 304
Hibakeresés terminálablak segítségével 304

Fejlesztési és terjesztési verziók, avagy a feltételes fordítás használata 306
Állítások (assertion) használata 308
Egy futtatható állomány vizsgálata az Object Browser segítségével 310
Az üzenetvezérlés elemzése 312
A WinSight használata 312
A Window Tree nézet: szülőablakok és gyerekablakok 313
Üzenetvezérlés és a Delphi események 316
Az elküldött üzenetek nyomon követése 317
Az alkalmazás a memóriában 321
A Windows rendszermemóriája 323
A szabad rendszermemória 324
Hogyan tovább? 326

Erőforrások használata 329
Windows erőforrások 329
Az erőforrás szerkesztők használata 330
Ikon vagy bittérkép felhasználása komponens tulajdonságaként 333
Az erőforrások közvetlen beolvasása 334
Programok és formok ikonjai 336
Többféle ikon használata a megváltozott állapotok jelzésére 337
A Tálca ikontartójának használata 339
A memóriaállapot megjelenítése az ikontartóban 340
A fő ablak elrejtése és megjelenítése 343
A tartóban lévő ikon elrejtése 345
A kurzor használata a Delphiben 346
Saját kurzor készítése 347
Változtatható kurzor 350
Stringtábla erőforrások használata 354
A stringek lefordítása idegen nyelvre 359
Verziószámok kezelése 361
Hogyan tovább? 364

Nyomtatás Delphi alkalmazásokból 367
Az egész form kinyomtatása 367
Saját nyomtatási párbeszédablak 368
A szabványos nyomtatás párbeszédablakok 369
A Printer objektum használata 371
Grafikák nyomtatási képe 372
A nyomtatási kód megosztása 376

Szöveg nyomtatása 378
Adatbázisrekordok és -táblák nyomtatása 381
A QuickReport komponensek 384
Egy gyors példa *385*
ReportSmith jelentések készítése 388
Saját jelentés készítése *390*
ReportSmith makró készítése *393*
Gyorsabban a ReportSmithmél *396*
Hogyan tovább? 398

Fájlműveletek az alkalmazások készítésénél ... 401
A fájlok és a Pascal nyelv 401
Szövegfájlok kezelése *402*
Egy szövegkonverter *406*
Általános adatok mentése *410*
Fájlkezelés a Delphi komponensekben 414
Fájlrendszer komponensek 414
Az adatfolyamok 417
Számfolyamok *419*
Komponensfolyamok *420*
Hogyan tovább? 424

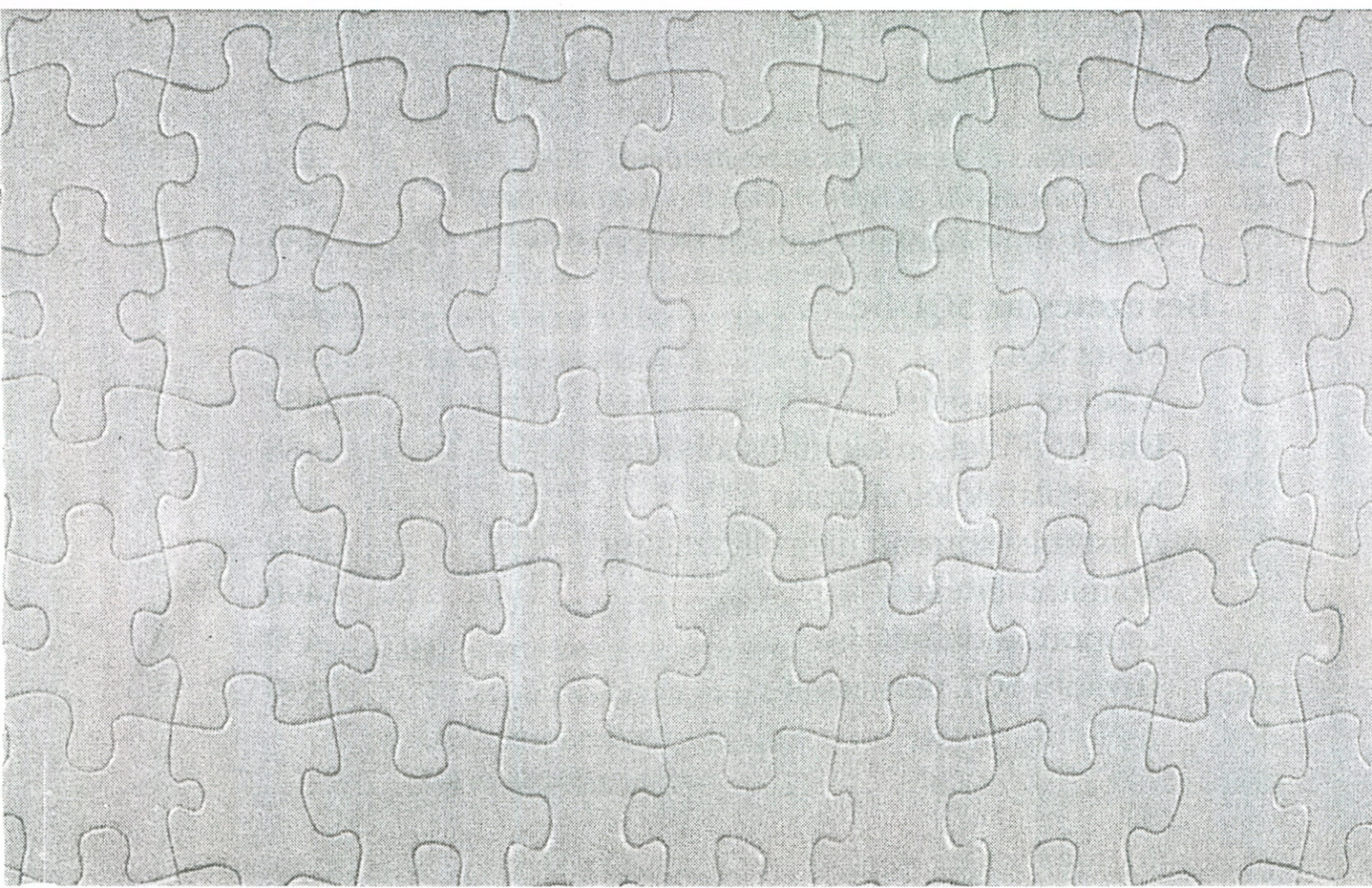
Adatcsere 427
Mi is a vágólap? 427
A Delphi vágólap-támogatása 429
Szöveg másolása és beillesztése *429*
Bitképek másolása és beillesztése *432*
Delphi komponensek bemásolása a vágólapra *435*
Speciális adat másolása a vágólapra *439*
Dynamic Data Exchange – technikai áttekintés ... 443
DDE párbeszédek *443*
A szerver és a kliens szerepe *443*
DDE a Delphiben 445
Egy egyszerű DDE példa *445*
Az egyszerű DDE szerver 445
Az egyszerű DDE kliens 446
DDE kapcsolat tervezési időben 447
DDE csatolások másolása és beillesztése 449
Kapcsolt adat másolása a vágólapra *449*
A DDE használata időzítők és grafikák esetében ... 450
Az automata szerver *450*

Grafikus DDE kliens *451*
Hogyan tovább? 453

Multimédiás lehetőségek 457
A Windows alapértelmezett hangjai 457
Minden ablaknak megvan a maga hangja *458*
A csipogástól a szimfóniákig *462*
Alapértelmezett hangok használata *463*
A Médialejátszó komponens 464
Hangfájlok lejátszása *465*
Videóklipek lejátszása *467*
Videóklip a formon *468*
A CD meghajtó használata 470
Utószó 473

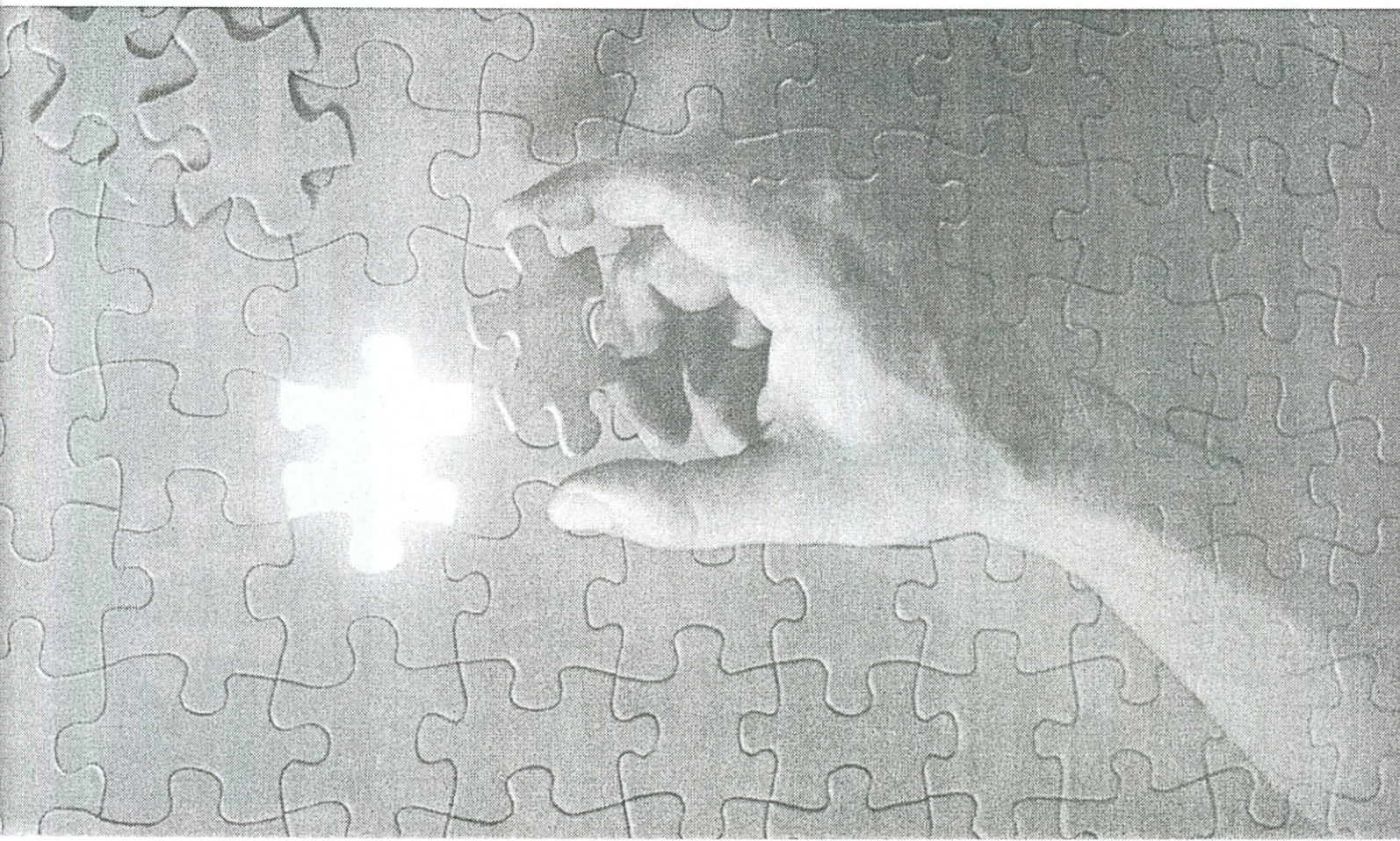
FÜGGELÉK 475
Az OOP szemlélet rövid áttekintése 479
Absztrakció a programozási nyelvekben 479
Osztályok 479
Öröklődés 480
Polimorfizmus 481
Az OOP egy definíciója 482
Az OOP nyelvek 482
Tiszta és hibrid OOP nyelvek *483*
Statikus és dinamikus típusellenőrzés *483*
Tradicionális és referenciaelvű objektummodell *483*
Objektum-orientált elemzés és tervezés 484

Bevezetés az SQL-be 487
Mi az SQL? 487
A Select utasítás 487
Ismétlődő rekordok kiküszöbölése 488
Kapcsolatok létrehozása 489
A listázási sorrend megváltoztatása 489
Számított értékek 490
Csoportok definiálása 490
További SQL utasítások 490



III. Rész

Komponensek és Könyvtárak



18. FEJEZET

Komponensek létrehozása

E fejezet tartalma:

- A VCL kibővítése
- Az első komponens elkészítése
- Új komponensek készítése
- Egy grafikus komponens építése lépésről lépésre
- Az óra komponens
- Tömbtulajdonság definiálása
- Egy nem vizuális komponens készítése

Komponensek létrehozása

Bár a Delphi programozók legtöbbje a létező komponenseket használja, néha szükség van saját komponensek készítésére, illetve a már létezők megváltoztatására. A Delphi egyik legvonzóbb tulajdonsága, hogy vele a komponenskészítés viszonylag egyszerű. Ezért ebben a fejezetben – bár ez a könyv Delphi alkalmazásokat fejlesztő programozók számára és nem komponenskészítőknek készült – tárgyaljuk a komponenskészítés összes mozzanatát, a következő fejezet pedig bemutatja, hogyan kell hozzáadni a már létező komponenseket a Delphihez.

A fejezetben előrehaladva áttekintést fogunk adni a Delphi komponenskészítésről, számos példán keresztül. Sajnos nem áll rendelkezésre elég hely bonyolult komponensek tárgyalására, de a fejezet minden információt meg fog adni ahhoz, hogy megismerjük a téma alapjait.

Megjegyzés

Sok további információ található a komponensek készítéséről a Delphi fejlesztői kézikönyvben (Delphi Developer's Handbook), beleértve az adatorientált komponenseket, és sok más egyéb technikát.

A VCL kibővítése

Ha új komponenst készítünk, minden egyes alkalommal a VCL egy osztályát terjesztjük ki. Ehhez az Object Pascal nyelv számos olyan tulajdonságát kell használni, amelyekre a komponensek használata közben ritkán van szükség. Ha nem teljesen világosak a használt Object Pascal nyelvi elemek, akkor a könyv első részében egy részletes leírás található a nyelvről, ahol minden felmerülő kérdésre választ kaphatunk. A 7. fejezet a VCL-ről ad egy bő áttekintést, annak hierarchiáját ábrázoló fával, a tulajdonságok, metódusok és események szerepének leírásával. Ha ezt a fejezetet kihagytuk, vagy bizonytalanak vagyunk a VCL alapjaival kapcsolatban, akkor olvassuk el ezt a részt a további ismerkedés előtt.

A Delphi komponensek tulajdonképpen osztályok, a VCL pedig nem más, mint azon osztályok összessége, amelyek a Delphi komponenseket leírják. Minden egyes alkalommal, amikor egy új csomagot (package) adunk hozzá a Delphihez néhány komponenssel, a VCL-t bővítjük egy új osztállyal. Az új osztály minden alkalommal valamely létező komponenshez tartozó osztály

leszármazottja lesz, új tulajdonságokkal, lehetőségekkel bővítve az eredeti osztálytól örökölt tulajdonságokat. Az új komponenst származtathatjuk úgynevezett absztrakt komponensosztályból is. Az absztrakt komponensek nem használhatók közvetlenül, belőlük csak új komponenseket hozunk létre. A VCL hierarchia számos ilyen komponenst kínál, megváltoztatva az eredeti viselkedésüket, tulajdonságaikat, teljesen új komponensek hozhatók létre. Az új komponens létrehozásakor a származtatott osztályban természetesen nem törölhetünk olyan tulajdonságot, amely az eredeti osztályban benne van. Ha a komponensosztály ilyen nem kívánt elemet tartalmaz, akkor ennek az eltávolítását egy magasabb szinten kell megtenni a VCL hierarchiában.

A komponensektől a csomagokig

A Delphi 1-ben és Delphi 2-ben úgy telepíthettünk új komponenst, hogy új egységeket (unit) adtunk hozzá, leírtuk az új osztályt és hozzáadtuk az új komponenst a komponens könyvtárfájlhoz (COMPLIB.DCL a Delphi 1-ben és CMPLIB32.DCL a Delhi 2-ben) Ezek a fájlok tárolták az összes tervezési időben használható komponenst. A komponens lefordított kódját a DCU fájlok tárolták, amelyeket a fordítás során szerkesztett be a fordító a kódba, természetesen csak azokét a komponensekéit, amelyeket az alkalmazás használt.

A Delphi 3-ban ez teljesen másképp történik. A komponensek úgynevezett komponenscsomagokban (component package) vannak. Minden komponenscsomag tulajdonképpen egy DLL, DPL (Delphi Component Library) kiterjesztéssel.

A csomagokhoz speciális fordítási opciók tartoznak. Két alapvető opció például a `$DESIGNONLY` és a `$RUNONLY`, amelyek a csomag viselkedését határozzák meg. Ezeket a kapcsolókat nem kötelező alkalmazni, csak a Delphi IDE használja őket. Ha egy csomagot próbálunk telepíteni, akkor az IDE ellenőrzi ezeket a kapcsolókat, és eldönti, hogy a felhasználó telepítheti-e ezt a csomagot, és hogy hozzá kell-e adni a csomagot a futásidejű komponensek listájához, avagy sem. A két kapcsolónak két lehetséges értéke van, így az összes lehetséges beállítási lehetőségek száma négy, két általánosabb és két speciális eset:

- Fejlesztési idejű csomagok: telepíthetők a Delphi fejlesztői környezetbe, ugyanaz a szerepük, mint a komponenskönyvtáraknak a Delphi előző verzióiban. A különbség az, hogy most egyszerre több komponenskönyvtárat, illetve csomagot tudunk telepíteni. A fejlesztési idejű csomagok tartalmazzák a komponens fejlesztési idejű részeit, úgymint tulajdonságszerkesztők és regisztrációs kód. Tartalmazhatják magukat a komponenseket is, de nem ez a javasolt megközelítés. Fontos, hogy a fejlesztési idejű komponensek statikusan szerkesztődnek be a futtatható állományba, ugyanúgy, mint az előző verzióknál.

- Csak futtatható csomagok: kizárólag Delphi alkalmazások futtatásakor használhatók. Nem telepíthetők a Delphi fejlesztői környezetébe, de automatikusan hozzáadódnak a futásidejű csomagok listájához, ha egy csomag telepítéséhez szükség van rájuk. A futásidejű csomagok rendszerint tartalmazzák a komponensosztály kódját, mert ezek szabadon terjeszthetők az alkalmazásokkal, de a Delphiben nem tudjuk őket új programok építésére felhasználni.
- Egyszerű komponenscsomagok (sem a fejlesztési idő, sem a futásidő kapcsoló nincs beállítva): nem telepíthetők és nem adódnak hozzá automatikusan a futásidejű csomagok listájához. Csak akkor van értelmük, amikor segédcsomagként egy másik csomag használja őket.
- Olyan csomagok, ahol mind a két kapcsoló be van állítva: telepíthetők és automatikusan hozzáadódnak a futásidejű csomagok listájához. Rendszerint ezek a csomagok olyan komponenseket tartalmaznak, amelyek kevés fejlesztési támogatást – vagy semennyit sem – igényelnek. Fontos, hogy az ezekkel készített alkalmazások felhasználói használhatják a csomagokat saját fejlesztéseikhez.

Tipp

A Delphi saját tervezés idejű csomagjainak nevei a DCL betűkkel kezdődnek (például DCLSTD30.DPL); a futásidejű csomagok nevei a VCL betűhármassal kezdődnek (például VCL30.DPL).

A 2. fejezetben megvizsgáltuk a csomagok hatását a futtatható állomány méretére vonatkozóan, most azonban koncentráljunk a csomagok készítésére, mivel ez a következő szükséges lépés a komponensek készítéshez, illetve telepítéséhez a Delphi 3-ban.

Ha lefordítunk egy futásidejű komponenst, létrejön egyrészt a dinamikusan szerkeszthető könyvtár (DPL fájl), valamint egy fájl a szimbóluminformációkkal (DCP fájl). Az utóbbi fájlt használja a Delphi fordító arra, hogy összegyűjtse a szimbólumokat azokról az egységekről, amelyeket a csomag tartalmaz, anélkül, hogy a DCU fájljaihoz hozzáférne. Ez csökkenti a fordítási időt, és lehetővé teszi csak a csomag terjesztését, az előre lefordított egységek nélkül. Az előre lefordított egységek mindig szükségesek ahhoz, hogy a komponensek statikusan szerkesztődjenek az alkalmazásba. Az előrefordított DCU fájlok (vagy a forráskód) terjesztésének ugyanakkor bizonyos komponensfajtáknál lehet értelme. Most pedig megvizsgáljuk, hogyan tudunk létrehozni egy csomagot, miután megvitattunk néhány ide vonatkozó alapelvet, és elkészítettük első komponensünket.

Megjegyzés

A DLL-ek futtatható állományok, függvényeket és osztályokat tartalmaznak, amelyek egy főprogramból vagy más DLL-ből futásidőben használhatók. A legfőbb előnyük, hogy sok alkalmazás használhatja ugyanazt a DLL-t, csak egy példányának kell a merevlemezen, illetve a memóriában lennie és így minden futtatható állomány mérete jelentős mértékben csökken. Ez a technológia érvényesül az új Delphi csomagoknál is. A 20. fejezetben részletesebben foglalkozunk a DLL-kel.

A komponensírás szabályai

Néhány általános szabály vonatkozik a komponensek készítésére. Részletes leírást a legtöbbjükről leginkább a Delphi komponenskészítők kézikönyvében (Delphi Component Writer's Guide) találhatunk. Ezt a leírást természetesen célszerű elolvasni, e könyvben lényegesen több információt találhatunk a témáról. Ezért fontos, hogy ha a komponenskészítés részleteibe akarunk látni, akkor használjuk a kézikönyvet. További segítséget találhatunk a Delphi Súgó fájljaiban, valamint a VCL forráskódjából és a VCL referencia-kézikönyvéből is sokat tanulhatunk (ezek megtalálhatók a Delphi Developer és Client/Server verziójában).

Álljanak itt az általam legfontosabbnak tartott szabályok a komponenskészítők számára:

- Tanulmányozzuk gondosan az Object Pascal nyelvet. Különösen figyeljünk az öröklődésre, a metódus felüldefiniálásra, a `public` és `published` rész közötti különbségre az osztályoknál, az események és tulajdonságok definíciójára.
- Tanulmányozzuk a VCL hierarchiájának felépítését és tartsuk kéznél a VCL hierarchia rajzát (ezt tartalmazza a Delphi is).
- Kövessük a Borland elnevezési konvencióit. Ezek közül számos a komponensekre vonatkozik, amint azt látni fogjuk később; ezeket a szabályokat követve megkönnyíthetjük azoknak a programozóknak a dolgát, akik majd a későbbiekben találkoznak komponensünkkel, és megkísérlik továbbfejleszteni azt.
- Ne készítsünk bonyolult komponenseket, másoljunk le bátran már létezőket és tartózkodjunk a függőségektől. Ez a három szabály tulajdonképpen azt jelenti, hogy a komponens felhasználója képes legyen

ugyanúgy használni a komponensünket, mint a Delphi eredeti komponenseit. Használjunk egyszerű tulajdonság-, metódus- és eseménynevekket, ha ez lehetséges. Ha a felhasználóknak nem kell bonyolult szabályokat megtanulniuk komponensünk használata közben (vagyis minimalizáljuk a metódusok és tulajdonságok közötti függőségeket) és egyszerűen hozzáférhetnek a tulajdonságokhoz beszédes nevekkel, akkor egészen biztosan elégedettek lesznek.

- Használjunk kivételeket. Ha valami rosszul működik, akkor a komponens generálhat egy kivételt. Ha bármely okból lefoglalunk valamilyen erőforrást, akkor azt egy `try-finally` blokkba kell zárni, és gondoskodni kell a destruktor hívásáról.
- Teszteljünk minden komponenst egy példaformon, mielőtt a telepítenénk. A telepítés közbeni súlyos hibák tönkretehetik a csomagot és a Delphit.
- A komponens befejezésekor adjunk hozzá egy bitképet, hogy ezzel a Delphi komponenspalettáján is meg tudjon jelenni. Ha a komponenst nem csak szűk körben szeretnénk terjeszteni, akkor célszerű azt Súgóval is ellátni.
- Készüljünk fel arra, hogy a kód írása közben nem kapunk vizuális támogatást, hacsak nem követjük az alábbi tanácsot:
- Használjunk egy komponenskészítő eszközt a fejlesztés felgyorsításához. A CD TOOLS könyvtárában egy ilyen eszköz található. Ez sokkal egyszerűbb, mint a professzionális változatok, de sok idő megtakarítható vele. Használata közben megtapasztalhatjuk, mekkora segítség egy ilyen eszköz a komponenskészítésnél.

Megjegyzés

Jelenleg két eszköz áll rendelkezésre Delphi komponensek készítésére, a **Component Create** a Potomac Document Software nevű cégtől, és a **Component Development Kit** (CDK) az Eagle Software-től. Mindkét program demováltozata megtalálható a CD-n.

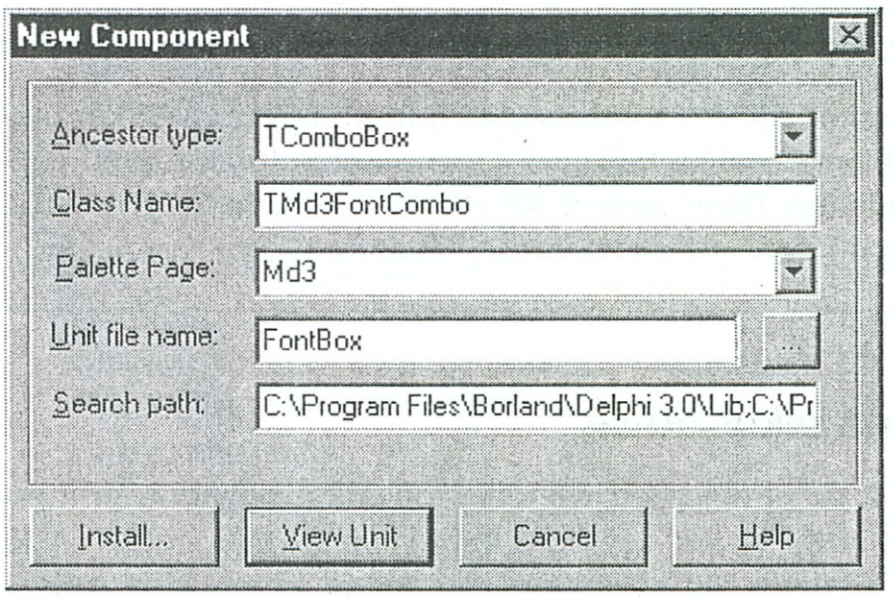
Az első komponens elkészítése

Mielőtt továbbhaladnánk az elméletben, és teljes példákat néznénk komponensekre, szeretnék mutatni egy-két egyszerű, majdnem triviális példát. A fejezet célja az, hogy a részletek figyelembevétele nélkül bemutassa, milyen egyszerű egy új komponens elkészítése és telepítése.

A Font kombinált lista

Számos alkalmazásban találkozhatunk olyan eszköztárral, amely egy betűtípusokat felsoroló kombinált listát tartalmaz. Található egy egyszerű példa erre (ComboBar) a 12. fejezetben. Ha sűrűn használunk ilyen testreszabott kombinált listát, akkor miért ne hoznánk létre erre a feladatra egy új komponenst? Ez valószínűleg kevesebb, mint egy percbe kerül. Először is zárjunk be minden aktív projektet a Delphi fejlesztői környezetében és indítsuk el a Komponens varázslót a **Component | New Component**, vagy a **File | New** választással. Utóbbinál nyissuk meg az Object Repository-t és válasszuk a Component-et a **New** oldalon. Bármelyik megközelítést használjuk, a Komponens varázslóhoz jutunk. A Komponens varázsló némileg különbözik az előző Delphi verziók Component Expertjétől. Ahogy az a 18.1. ábrán is látszik, a varázsló számára a következő információkat kell megadni:

- A szülőtípus neve, ez annak a komponensosztálynak a neve, ahonnan a komponensünket származtatni fogjuk. Ebben az esetben mi a `TMd3ComboBox`-ot használjuk.
- Az új komponens osztályának neve, mi a `TFontCombo` nevet válasszuk.
- A komponenspalettán annak az oldalnak a neve, ahol a komponensünket látni szeretnénk, amely lehet egy már létező oldal, vagy egy új is. Mi most hozzunk létre egy új oldalt, amit nevezzünk Md3-nak.
- Annak a Pascal egységnek a fájlneve, amelyben a Delphi elhelyezi az új komponens forrását, ide írjunk `FontBox`-ot.
- Az aktuális keresési útvonalat.



18.1. ábra *Az új `Tmd3FontCombo` komponens a Komponens varázslóban*

Kattintsunk a **View Unit** gombra, és a Komponens varázsló készít egy egyszerű Pascal kódot a komponensünk struktúrájával. A másik gomb (Install) a komponens telepítésére szolgál valamely csomagban. Én először a kódot szeretném megmutatni, és csak azután beszélek a telepítésről:

```
unit FontBox;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

Type
TMd3FontCombo = class(TComboBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Md3', [TMd3FontCombo]);
end;

end.
```

Az egyik kulcseleme ennek a forrásnak az osztálydefiníció, amely a szülőosztály megadásával kezdődik. A másik fontos rész a `Register` eljárás. Tényleges munkát, mint láthatjuk, a Komponens varázsló nem nagyon végez.

Most neki is kezdhetünk a kódírásnak, mindenféle külső eszköz nélkül. Természetesen jelen esetben a kód nagyon egyszerű lesz. Csak arra lesz szükségünk, hogy az Items tulajdonságot inicializáláskor feltöltsük a rendszerben található betűtípusok neveivel. Ahhoz, hogy ezt megtehessük, felül kell definiálnunk a Create metódust az osztálydeklarációban, hozzáadva a következő utasítást: Items:= Screen.Fonts. Bár megpróbálhatjuk a komponensünket hozzáadni a komponenskönyvtárhoz, az nem lesz használható. A probléma az,

hogy nem férünk hozzá a kombinált lista Items tulajdonságához (ez azokat a stringeket tartalmazza, amelyek a listában jelennek meg), mielőtt a komponens teljesen létre nem jön. Emiatt néhány fontos tulajdonságot (mint például a Parent tulajdonságot, amely arra a formra hivatkozik, amely a komponenst tartalmazza) nem tudunk pontosan beállítani, míg a komponens létrehozása le nem fut.

Tipp

Használjunk elnevezési konvenciókat komponenskészítés közben. Minden Delphibe telepített komponensnek más osztálynévvel kell rendelkeznie, ezért aztán a legtöbb Delphi komponensfejlesztő azt választja, hogy két vagy három karakterből álló jelzőprefixeket (kezdőkaraktereket) tesz a komponensnevek elé. Én is hasonlóan jártam el, amikor az Md3 betűcsoportot használtam a könyvben bemutatott komponensek azonosítására. Ennek a megközelítésnek nagy előnye, hogy telepíthetjük a Tmd3FontCombo komponenst akkor is, ha egy TFontCombo komponenst már előzőleg telepítettünk.

Ahelyett, hogy a Create konstruktorban adnánk meg az új stringeket, ezt a műveletet a CreateWnd eljárásban hajtjuk végre, amely a komponens létrehozása után fut le, és létrehozza annak ablakát. A Create metódus átírását kihagyhattuk volna, és rögtön a CreateWnd-vel kezdhettük volna, de úgy döntöttem, hogy fontos látnunk a kettő közötti különbséget. Most pedig lássuk az új osztálydefiníciót:

```
type
  TMd3FontCombo = class(TComboBox)
  public
    constructor Create (AOwner: TComponent); override;
    procedure CreateWnd; override;
  published
    property Style default csDropDownList;
  end;
```

Íme a forráskód a két új metódussal (a teljes program megtalálható a CD FontBox mappájában):

```
constructor TMd3FontCombo.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  Style := csDropDownList;
end;
```

```
procedure TMd3FontCombo.CreateWnd;
begin
  inherited CreateWnd;
  Items := Screen.Fonts;
end;
```

Vegyük észre, hogy a komponens Style tulajdonságának Create metóduson belüli beállításán kívül ezt a tulajdonságot a `default` kulcsszóval is újradefiniáltuk. Meg kell tennünk mind a kettőt, mert a `default` kulcsszó használata a tulajdonság deklarációjánál nincs hatással az inicializáló értékre. Akkor mi értelme van az alapértelmezett érték beállításának? Nos az, hogy a `default` kulcsszó mondja meg, hogy a komponens inicializációs kódja állítja majd be a tulajdonság értékét.

Csomagok létrehozása

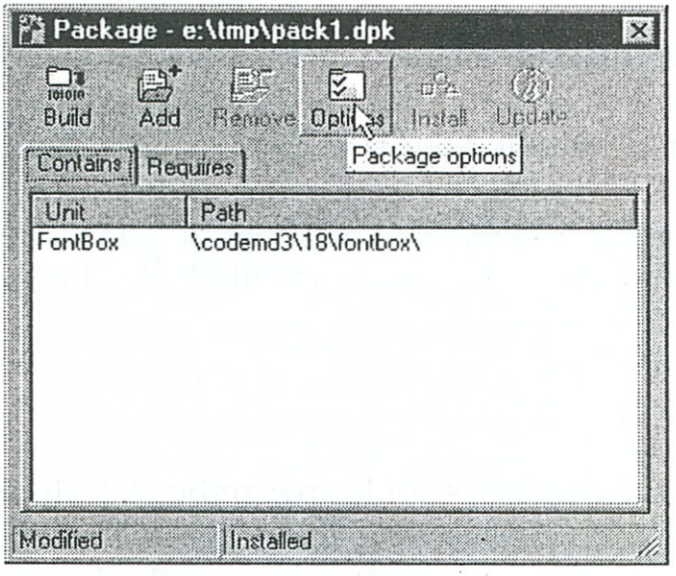
Most pedig telepítenünk kell a komponenst a fejlesztői környezetbe. A Delphi 3-ban a komponenseket mindig csomagokban telepítjük. Ebben a példában mi egy új csomagot fogunk készíteni, vagy pedig egy már létezőt használunk, például az alapértelmezett `users` `package`-et, mint ahogy azt a 6. fejezetben tettük. Új csomag létrehozása mindenesetre nagyon egyszerű.

Bármelyik módot választjuk is, menjünk a Component/Install Component menüpontra. Az itt kapott dialógusablakban egy oldalon a már létező csomagokat látjuk, és találunk itt egy másik oldalt, ahol új csomagot hozhatunk létre. Ha ezt választjuk, akkor egyszerűen be kell írnunk egy fájlnevet, valamint a csomag leírását. Ha az OK gombot megnyomjuk, akkor belépünk a Csomagszerkesztőbe (Package Editor, lásd a 18.2. ábrát), amelynek szintén két oldala van:

- A **Contains** oldal a csomagban található komponensek listáját tartalmazza (precízebben megfogalmazva, azokat az egységeket, amelyek a csomag komponenseit írják le.)
- A **Requires** oldal a csomaghoz szükséges további csomagok listáját tartalmazza. Általában csak a vcl30 csomagra van szükség (ez a legfontosabb futásidejű elem, amely a Delphi VCL alapvető részeit tartalmazza), de szükség lehet a vcldb30 csomagra is (ebben találjuk az adatbáziskezeléssel kapcsolatos osztályokat).

Ha a komponenst egy új csomaghoz adjuk hozzá, akkor azt egyszerűen csak le kell fordítani és telepíteni (használjuk a két megfelelő gombot az eszköztárból a csomagszerkesztőben), és azonnal látjuk az új komponenst az Md3 oldalon, a komponenspalettán. A komponens Register eljárása mondja meg a Delphinek, hova kell elhelyeznie az új komponenst. Alapértelmezésben a szülőosztály bitképével kerül be, mert nem gondoskodtunk sajátról (a továbbiakban

ezt is megtesszük az újabb példák során). Vegyük észre, hogyha az egérkurzort az új komponens bitképe fölé visszük a komponenspalettán, akkor tippként megjelenik az osztály neve, a kezdő T betű nélkül.



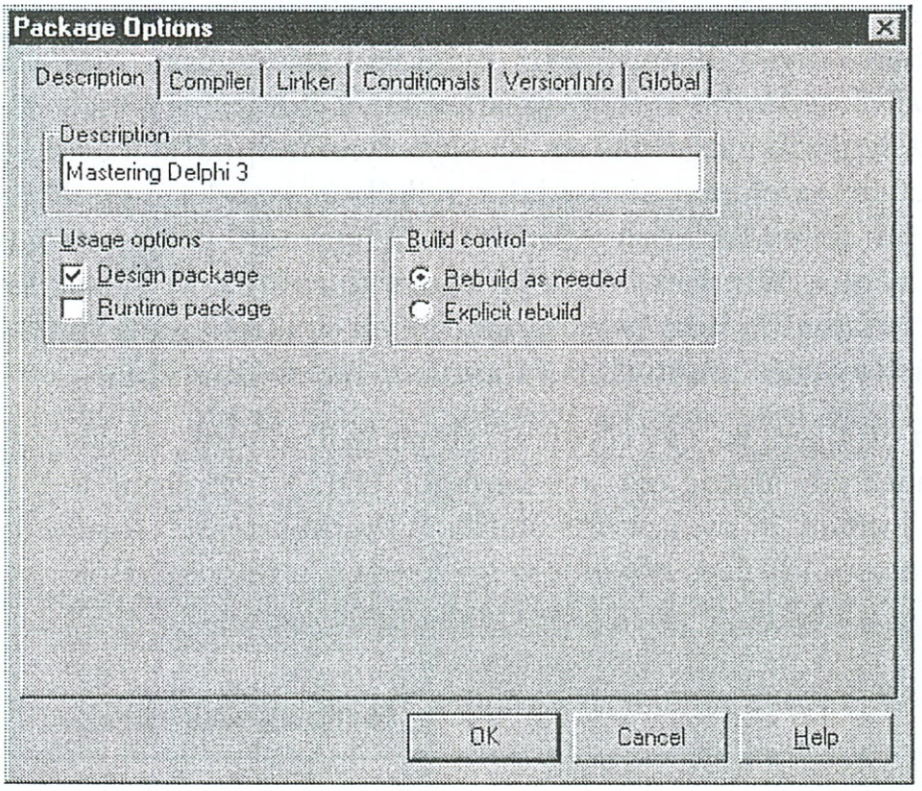
18.2. ábra *A Delphi csomagszerkesztője*

Hogyan épül fel egy csomag?

Mi történik a háttérben, amikor lefordítjuk a csomagot? A csomagszerkesztő egyszerűen létrehozza a forráskódot a csomag projekt számára: ez egy speciálisan fordított DLL lesz a Delphiben. A csomag projektek DPK (Delphi PacKage) kiterjesztéssel kerülnek mentésre. Egy általános projekt forrása a következőképpen néz ki:

```
package Md3Pack;

{$R *.RES}
{$ALIGN ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
...
{$DESCRIPTION 'Mastering Delphi 3 Package'}
{$DESIGNONLY}
{$IMPLICITBUILD ON}

requires
  vcl30;
contains
  FontBox;
end.
```

Mint látható, a Delphi speciális új kulcsszavakat használ a csomagokhoz: az első a `package` kulcsszó (hasonló a `library` kulcsszóhoz, amelyről a 20. fejezetben lesz szó). Ez a kulcsszó vezeti be az új csomag projektet. Ezután következik a fordítási opciók listája. Most csak a lista egy részét tekintjük át: a csomagok nem használnak külső opciós fájlt, mindent tartalmaznak a forráskódjukban. A fordítási direktívák között található a `$DESCRIPTION`, amely azt a leírást tartalmazza, amely a Delphi fejlesztői környezetében látható. Azaz, ha telepítjük az új csomagot, akkor ez a leírás megjelenik a Project Options párbeszédablak **Packages** oldalán, ha meg akarjuk nézni, akkor a **Component | Install Packages** menüpontot kell kiválasztanunk. Ez a párbeszédablak látható a 18.3. ábrán.



18.3. ábra *A csomagok projekt opciói. Láthatjuk az új komponenst, amit az előbb hoztunk létre*

Az általános direktívák mellett, mint amilyen a `DESCRIPTION`, vannak még mások is a csomagok opcióinak beállítására. Ezek legtöbbje könnyen elérhető a Csomagszerkesztőben, ha az **Options** gombot megnyomjuk. Az opciók listája után következik a `requires` és a `contains` kulcsszó, amelyek azokat az elemeket tartalmazzák, amelyek vizuálisan megjelennek a csomagszerkesztő két oldalán. Csupán emlékeztetőül: az első lista a csomaghoz szükséges csomagokat tartalmazza, a másodikban pedig a csomaghoz tartozó egységek vannak.

Mi történik a csomag készítésekor? A DPK forrásfájl mellet a Delphi létrehoz egy DPL fájlt is, amely a csomag dinamikusan szerkesztett változata, valamint egy DCP fájlt a szimbóluminformációkkal. A DCP fájl a csomagban található egységek DCU fájljainak szimbóluminformációit tartalmazza.

A DPL és DCP fájl szükséges csak ahhoz, hogy a csomagot terjeszteni tudjuk (bár a megfelelő DCU fájlokat is hozzáadhatjuk). Ezek a DCU fájlok akkor szükségesek, ha statikusan akarjuk a csomag komponenseit egy alkalmazáshoz kapcsolni.

A fejezet komponenseinek telepítése

Most, miután lefordítottuk az első csomagunkat, megkezdhetjük a használatát. Mielőtt ezt megtennénk, meg kell említenem, hogy az Md3Pack csomagot én kiterjesztettem és minden olyan komponens megtalálható benne, amelyet ebben a fejezetben hozunk létre, beleértve ugyanannak a komponensnek különböző változatait is. A legjobb megoldás az, ha egy olyan mappába másoljuk a csomagot, ami benne van az elérési útvonalak egyikében, ezzel a megoldással elérjük, hogy a Delphi fejlesztői környezet és az őt használó programok is lássák. Alapértelmezésként a Windows System könyvtára tűnik jó választásnak, bár ez idővel egyre nagyobb lesz és hatalmas helyet szakít ki merevlemezünkből. Célszerű tehát egy másik könyvtár választása, ha ez lehetséges.

Emlékeztetőül: minden esetben, ha egy olyan alkalmazás fordítunk, amely egy futásidejű csomagot használ, akkor ezt a csomagot (DLL) telepíteni kell a kliensgépen is. Ha e helyett a programot statikus szerkesztéssel fordítjuk, akkor a DLL csak a fejlesztői környezetben lesz szükséges, a felhasználók számára nem kell telepítenünk az alkalmazás futtatásához.

Megjegyzés

Egyszeri csomagkészítés és telepítés mellett a Delphi csomagcsoportokat is tud kezelni. A Package Collection Editor (PCE.EXE, a Delphi BIN könyvtárában) lehetővé teszi több csomag elhelyezését egy DPC (Delphi Package Collection) fájlba. Ezt a fájlt aztán ugyanúgy telepíthetjük a Delphi alá, mint egy csomagot.

A Font kombinált lista használata

Nos, most már készen állunk arra, hogy egy olyan Delphi tesztprogramot írjunk, amely használja kész komponensünket. Menjünk a komponenspalettára és válasszuk ki az új komponenst, majd helyezzük el a formon. Az új komponens megjelenése teljesen azonos a régi kombinált listáéval. Ha megnézzük az

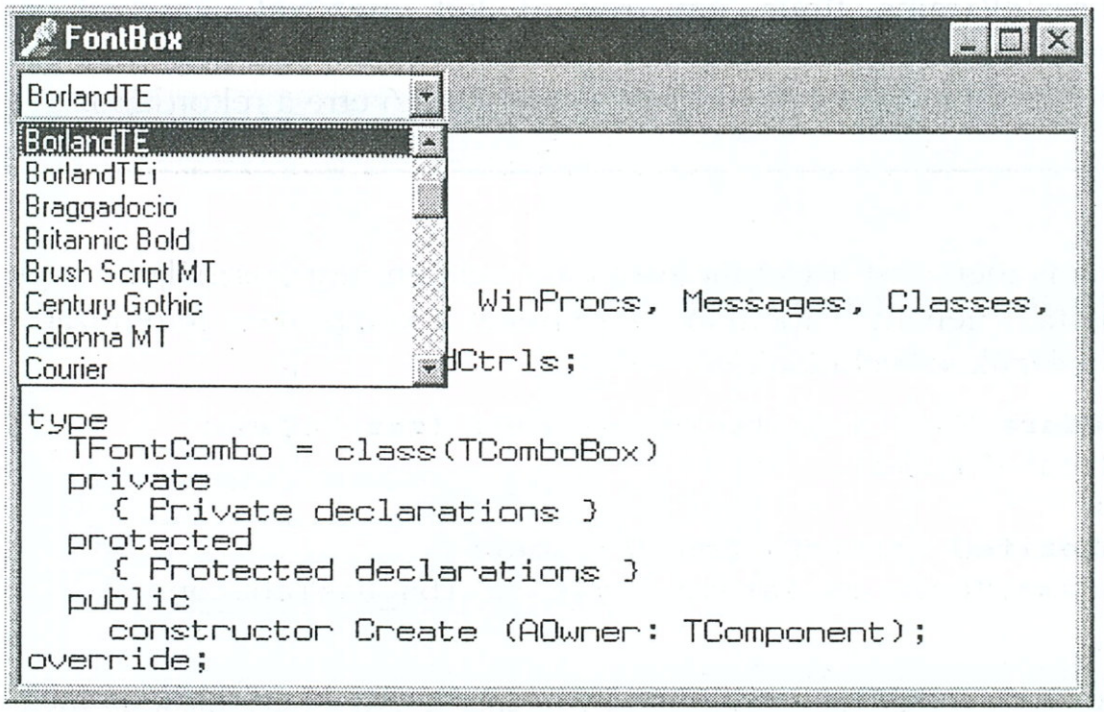
Object Inspectorban az Items tulajdonságot, akkor ott a számítógépünkön telepített betűtípusok listáját találjuk. Készítsünk egy egyszerű példaprogramot: tegyünk a formra egy Memo komponenst és írjunk bele egy tetszőleges szöveget. A programban kevés kódot kell írnunk. Ha a felhasználó kiválaszt egy új betűtípust a kombinált listából, akkor ezzel a betűtípussal jelenítjük meg a Memoba írt szöveget.

```
procedure TForm1.Md3FontCombo1Change(Sender: TObject);
begin
  Memo1.Font.Name :=
    Md3FontCombo1.Items [Md3FontCombo1.ItemIndex];
end;
```

A program indulásakor a kombinált lista aktuális betűtípusát használja a Memo:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Md3FontCombo1.ItemIndex :=
    Md3FontCombo1.Items.IndexOf (Memo1.Font.Name);
end;
```

E rövid program célja az, hogy egyszerű példát mutasson az új komponensünk viselkedésére, amelyet az imént készítettünk (18.4. ábra). A komponens még nem túl jól használható – ugyanezt a hatást elérhetjük hagyományos komponensekkel és pár sor kóddal – de az ilyen egyszerű komponensek készítése adhat a későbbiekben nagyon hasznos ötleteket.



18.4. ábra *A DemoFb példaprogram*

Tabulált listapanel készítése

A Delphi `ListBox` komponense nem támogat egy a Windowsban nagyon hasznos dolgot: a tabulátorkarakterek használatát. Ezáltal az adatok táblázatos formában történő megjelenítése nehézkes. Könnyen készíthetünk egy új komponenst, amely rendelkezik ezzel a képességgel. Mivel az oszlopok kezelése nem egyszerű feladat (mint azt később a fejezet során látjuk majd), ezért a Borland döntése nem véletlen. A listapanelek viselkedésének kiterjesztése helyett a Borland programozói egy radikálisabb megoldást választottak: a `Grid` és `ListView` komponensek családjának létrehozását. A célom ezzel a példával most is csak az, hogy bemutassam, hogyan hozunk létre egy új komponenst.

A listapanel csak akkor fogadja el a tabulátorkaraktereket, ha a létrehozásánál az `lbs_UseTabStops` ablakstílust használjuk. Tehát szükségünk lesz egy módszerre, amellyel megváltoztathatjuk a kontrol létrehozásának módját. Már láttuk egy előbbi példában, hogy az ablak a létrehozandó komponenshez csak a második lépésben kapcsolódik, a CreateWnd metódussal. Ha ez a metódus lefutott, akkor megvizsgálhatjuk a CreateWindow API függvény paramétereit és megváltoztathatjuk őket, mielőtt az aktuális ablaknak a hely lefoglalásra kerül. Ezt a CreateParams metódus újradefiniálásával tehetjük meg, amelynek paramétere TCreateParams típusú.

Megjegyzés

Tapasztalt Windows programozók talán már tudják, hogy a TCreateParams típus egy rekord két mezővel, amelyek a CreateWindow API függvény paramétereinek felelnek meg. A `wm_Create` üzenet egyik paramétere mutató erre a rekordtípusra.

A CreateParams metódust a Delphi komponensekben arra használjuk, hogy megváltoztassunk néhány alapértelmezett értéket a komponens létrehozásakor. Például nézzük az alábbi kódot:

```
procedure TMd3TabList.CreateParams (var Params:
  TCreateParams);
begin
  inherited CreateParams (Params);
  Params.Style := Params.Style or lbs_UseTabStops;
end;
```

A program először meghívja a szülőosztály CreateParams metódusát és megadja az alapértelmezett értékeket. Ezután beállítjuk az `lbs_UseTabStops` kapcsolót ablakstílusnak. Mostantól használhatjuk a tabulátor karaktereket a listapanelben, s így megjeleníthetünk oszlopokat. Indítsuk el a Komponens

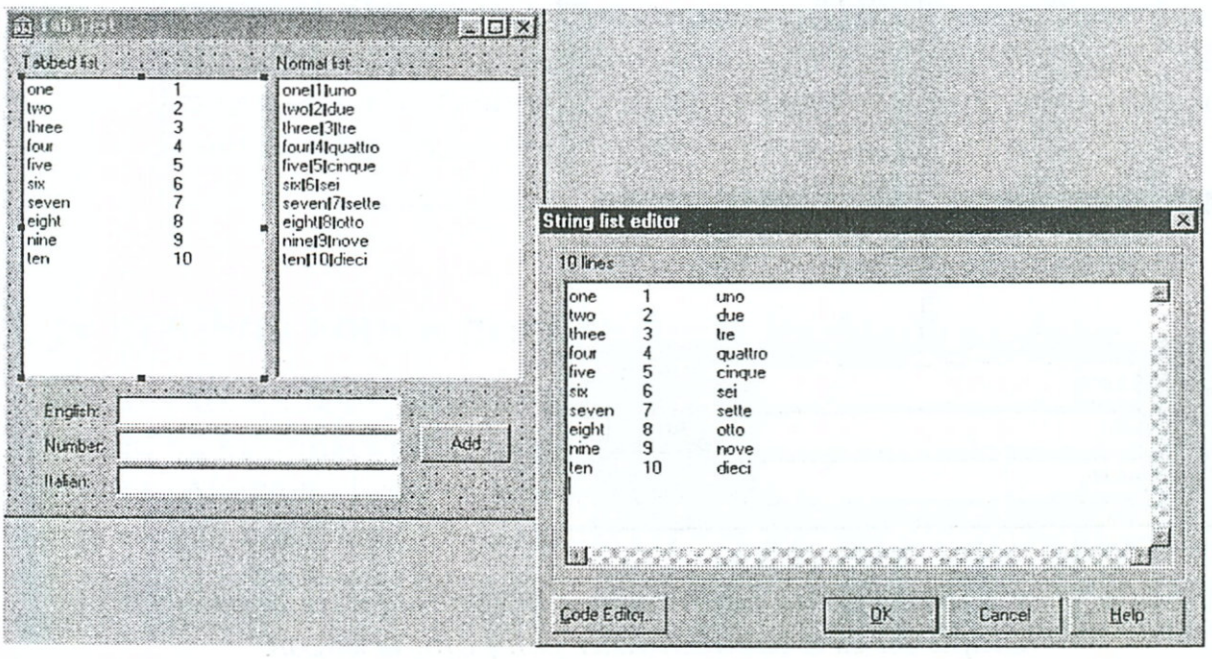
varázslót, és nevezzük el az új komponensosztályt `TMd3TabList`-nek, a szülőosztály legyen a `TListBox`, a paletta lapjai közül pedig válasszuk az Md3-at. Az automatikusan létrehozott Pascal forrásból kitöröltem néhány felesleges deklarációt és beleírtam a fenti metódust, és természetesen a deklarációja belekerült az osztály definíciójába:

```
type
  TMd3TabList = class(TListBox)
  public
    procedure CreateParams (
      var Params: TCreateParams); override;
  end;
```

A példánk kódja egy újabb példája annak, hogy a komponensírás nem egy ördöngős feladat. Az elkészült komponens kipróbálására készítettem egy példaprogramot, amely a TabTest névre hallgat és a CD TabList könyvtárában található.

A készített listapanel komponens kipróbálása

Az új Md3TabList komponens kipróbálásához hozzunk létre egy új projektet, és a formra helyezzünk el egy `Md3TabList`, valamint egy `ListBox` komponenst egymás mellé. Legyen a formon három szerkesztőmező és egy gomb is, amelyet az új stringek hozzáadására tudunk használni. Miután ez megtörtént, írjunk be néhány stringet mindkét listába, ugyanazokat a sorokat használjuk, és tegyünk bele tabulátor karaktereket is. A 18.5. ábrán láthatjuk az általam beírt sorokat. A háttérben látható, hogyan jelenítette meg a két listapanel a sorokat.



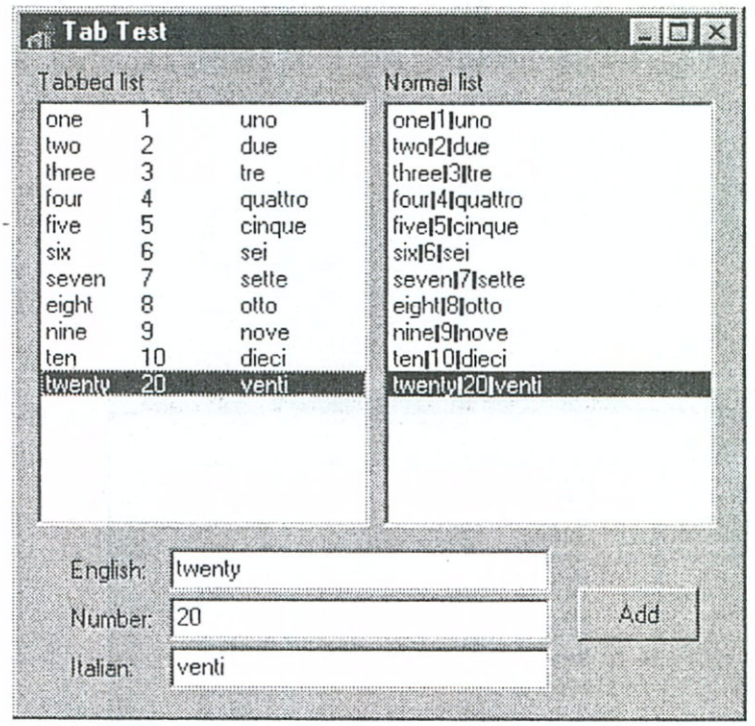
18.5. ábra *A TabTest példa formja tervezési időben*

Tabulátor karakterek elhelyezéséhez a String List tulajdonságszerkesztőjében nyomjuk le egyszerűen a CTRL+TAB kombinációt a TAB billentyű helyett.

Ha futtatjuk ezt a programot, hogy lássuk a két komponens viselkedése közti különbséget, működés közben új sorokat is hozzáadhatunk a listákhoz. A három szerkesztőmezőbe írhatjuk a szöveget, a szövegrészek közé a program tabulátor karaktereket helyez el. A hozott új stringet kísérli meg a program hozzáadni mindkét listapanelhez, ezt a műveletet az alábbi kód végzi:

```
procedure TForm1.AddButtonClick(Sender: TObject);
var
  NewItem: String;
begin
  if Edit1.Text <> '' then
  begin
    NewItem := Edit1.Text + #9 + Edit2.Text +
      #9 + Edit3.Text;
    Md3TabList1.Items.Add (NewItem);
    ListBox1.Items.Add (NewItem);
  end;
end;
```

A kód hatása a 18.6. ábrán látható, ahol a 20-as számot adtam hozzá arab és római számokkal a listapanelekhez. A tesztelő program ezen kívül más kódot nem tartalmaz.



18.6. ábra *A TabTest példa bemutatja a különbséget az eredeti Delphi listapanel és a tabulált lista komponens között*

Új komponensek készítése

Azok a komponensek, amelyeket eddig készítettünk, nagyon kis mértékben tértek el az eredeti komponensektől, amelyekből származtattuk őket. Az esetek nagy részében csak ilyenekre lesz szükségünk. Más esetekben viszont szükség van vadonatúj komponensekre, amelyeket a `TCustomXXX` általános osztályokból származtatunk. Ebben az esetben az alábbi absztrakt osztályokból választhatunk:

- `TWinControl`: Az ablakra épülő komponensek szülőosztálya. Az ebből az osztályból származó komponensek birtokolhatják a fókuszt, és üzeneteket fogadhatnak az operációs rendszertől. Lekérdezhetjük az ablakkezelőjét API függvény hívásával.
- `TGraphicControl`: Olyan vizuális komponensek szülőosztálya, amelyeknek nincs ablakkezelőjük. Ezek a komponensek nem kaphatják meg a fókuszt, és nem kaphatnak direkt módon üzeneteket sem a rendszertől.
- `TComponent`: Minden komponens szülőosztálya, és közvetlen szülőosztály a nem vizuális komponensek számára.

A fejezet hátralévő részében különböző szülőosztályok segítségével készítünk komponenseket és megvizsgáljuk a köztük lévő különbségeket.

Megjegyzés

Ha egy új komponenst származtatunk a VCL hierarchia magasabb szintjéről, akkor a létrejövő komponens rendelkezik olyan tulajdonságokkal, amelyek minden komponensnél megvannak. Nézzük meg a 7.7. ábrát, ahol a magas szintű VCL osztályok néhány tulajdonsága látható.

Egy grafikus komponens építése lépésről-lépésre

A grafikus komponens, amelyet készítünk, egy nyíl alakzat lesz. Felhasználhatjuk majd az információ irányának jelölésére, vagy valamilyen művelet végrehajtására. Ahelyett, hogy a komponens végleges változatát tekintenénk át, amely eléggé komplex, inkább következetes lépések sorozatán keresztül nézzük végig a felépítését. Egy olyan programmal fogjuk tesztelni a komponensünket, amely mindig futásidőben hozza létre azt, hogy elkerüljük az őt tartalmazó csomag sorozatos újrafordítását (bár ez a folyamat a csomagokkal a Delphi 3-ban jelentősen felgyorsult a Delphi 2-ben alkalmazott módszerhez

képest, ahol az egész komponenskönyvtárat frissíteni kellett). A komponens, amit az Md3Pack csomaghoz hozzáadunk, már a végleges változat lesz. Ahogy a komponens építésében előrehaladunk, úgy ismerjük meg sorra a legfontosabb elveket:

- Az új felsorolás típusú tulajdonságokat a felsorolás típusú adattípusokból származtatjuk
- A `TPersistent` származtatott osztályok (úgymint `TPen` és `TBrush`) tulajdonságait, valamint a létrehozáshoz és megszüntetéshez kapcsolódókat, és az OnChange eseményeket komponensünk belsejében kell kezelnünk.
- A Paint metódus implementációja, amely a komponens felhasználói felületét határozza meg, és amelynek elég általánosnak kell ahhoz lennie, hogy a különböző tulajdonságok minden lehetséges értékét lekezelje, beleértve a Width és Height tulajdonságokat is. A Paint metódus fontos szerepet tölt be a grafikus komponensekben.
- Teljesen új eseménykezelőt kell írni, felkészülve a jellegzetes műveletekre, amiket a felhasználó végrehajthat (duplakattintás a nyílon). Ez azt jelenti, hogy közvetlenül kell kezelnünk a Windows üzeneteket és a Windows API grafikus területfüggvényeit.
- A tesztprogramot a komponenssel együtt folyamatosan fejlesztjük, lépésről-lépésre.
- Képet készítünk a komponenshez a palettára.

Megjegyzés

Tesztprogramok készítése komponensekhez teljesen általános és nagyon hasznos. Ezért néhány fejlesztő kínál komponenstesztelő programokat, melyek jól jöhetnek, ha egy bonyolultabb komponenst fejlesztünk. Ezek közül a leginkább ismert a ReAct az Eagle Software-től, demó verziója megtalálható a CD-n.

Felsorolás típusú tulajdonság definiálása

Miután a Komponens varázslóban a `TGraphicControl`ból származtattunk egy új osztályt, elkezdhetjük testreszabni komponensünket. A nyíl négy irányba mutathat: föl, le, jobbra és balra. A lehetőségek kifejezésére definiálunk egy felsorolási típust:

```
TArrowDirection = (adUp, adDown, adLeft, adRight);
```

Ez a felsorolási típus definiál egy privát adattagot a komponensben, és paraméterként szerepel egy eljárásban, amelyet a megváltoztatására használunk:

```
TMd3Arrow = class(TGraphicControl)
  private
    FDirection: TArrowDirection;
    procedure SetDirection (Value: TArrowDirection);
    ...
```

Most, hogy már a `private` részben van egy adattagunk és egy metódusunk a beállítására, létrehozhatjuk a megfelelő tulajdonságot:

```
published
  property Direction: TArrowDirection read FDirection
      write SetDirection default adRight;
```

A tulajdonság típusa `TArrowDirection`, amely közvetlenül az FDirection mezőből veszi az értékét, és a SetDirection metódussal tudunk új értéket írni bele. Az alapértelmezett érték az adRight érték. Szükségünk lesz egy olyan eljárásra, amely frissíti a nyíl irányát, azaz újrarajzolja, amikor a tulajdonság megváltozik:

```
procedure TMd3Arrow.SetDirection (Value:
  TArrowDirection);
begin
  if FDirection <> Value then
  begin
    FDirection := Value;
    Invalidate;
  end;
end;
```

Figyeljük meg, a változás átvezetésére csak akkor van szükség, ha ténylegesen változott a tulajdonság értéke. Ellenkező esetben az egész eljárás belső magja kimarad, és a metódus végrehajtása befejeződik. Ez a módszer elég általános és ezt fogjuk alkalmazni a tulajdonságok beállításánál.

Nem szabad elfelejtenünk az alapértelmezett érték megadását a komponens konstruktorában:

```
constructor TMd3Arrow.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDirection := adRight;
end;
```

Mint már említettük, a tulajdonság alapértelmezett értékének beállítása a deklarációnál csak arra való, hogy meghatározzuk, milyen módon lesz elmentve a tulajdonság értéke. A Create konstruktor az új komponens típusdefiníciójának `public` részében, `override` kulcsszóval van definiálva. Ez a kulcsszó

nagyon fontos, mert ha nincs ott, akkor a Delphi az új komponens létrehozásakor nem a saját konstruktorát fogja meghívni, hanem a szülőosztályét. Így néz ki az osztály deklarációja:

```
type
  TArrowDirection = (adUp, adDown, adLeft, adRight);
  TMd3Arrow = class(TGraphicControl)
  private
    FDirection: TArrowDirection;
    procedure SetDirection (Value: TArrowDirection);
  protected
    procedure Paint; override;
  public
    constructor Create (AOwner: TComponent); override;
  published
    property Direction: TArrowDirection read FDirection
        write SetDirection default adRight;
  end;
```

Tulajdonság elnevezési konvenciók

Az Arrow komponens definíciójában már részleteztük az elnevezési konvenciókat tulajdonságokra, metódusokra és mezőkre. Ezek összegzése:

- A tulajdonság neve legyen olvasható és beszédes
- Ha egy privát adatmezőt tulajdonság értékének tárolására használunk, akkor a neve kezdődjön F (field) betűvel, folytatva a megfelelő tulajdonság nevével.
- Ha egy metódust egy tulajdonság értékének megváltoztatására használunk, akkor a neve kezdődjön a Set szóval és folytatódjék a megfelelő tulajdonság nevével.
- A tulajdonság értékét olvasó függvény neve kezdődjön a Get szóval, és ismét csak folytatódjon a megfelelő tulajdonság nevével.

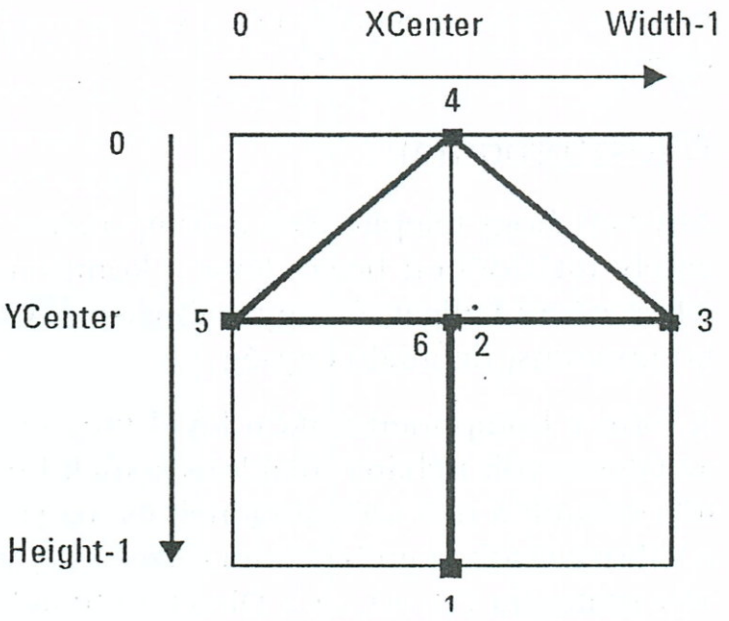
Ezáltal a programjaink sokkal olvashatóbbak lesznek, bár a fordító nem követeli meg ezek betartását. Ezek a konvenciók részletesen le vannak írva a Delphi komponenskészítők kézikönyvében (Delphi Component Writers Guide) és ezeket támogatják a komponenskészítő eszközök is, például az általam készített, egyszerű programocska.

A Paint metódus

A nyíl megrajzolása a feladat egyik bonyolult része. Elidőzünk egy kicsit a dologgal, hogy megtanuljunk egy-két dolgot a Canvas tulajdonság használatáról, valamint a Windowsban elérhető GDI (Graphics Device Interface) API hívásokról. Azért foglalkozunk ezzel olyan sokat, mert elég ritkán fordul elő, hogy egy Delphi programban közvetlenül a képernyőre rajzolunk, általánosabb, hogy ez a komponenseken belül zajlik le. A komponenskészítőknek sokkal jártasabbaknak kell lenniük a grafikus kimenet kezelésében, mint a többi Delphi programozónak.

Ebben a példában, mint a grafikus komponensek legtöbbjében, felül kell definiálnunk a Paint metódust a definíció `protected` részében (mint ahogy az a lenti forráskódban látszik) és a Canvas tulajdonságot is fogjuk használni (automatikusan elérhető minden grafikus komponensben). A MoveTo és a LineTo metódusok megfelelő hívásaival megrajzolhatjuk a különböző irányú nyilakat, mint ahogy azt a 19.7. ábra szemlélteti a felfelé mutató nyíllal.

```
1) MoveTo (XCenter, Height-1);
2) LineTo (XCenter, YCenter);
3) LineTo (Width 1, YCenter);
4) LineTo (XCenter, 0);
5) LineTo (0, YCenter);
6) LineTo (XCenter, YCenter);
```



18.7. ábra *A felfelé mutató nyíl kódja és annak eredménye*

A program kiszámítja a komponens területének középpontját, és ezt a pontot használja fel a nyíl pozíciójának meghatározásához, azaz annak két részének megrajzolásához (egy vonal és egy háromszög). Figyeljük meg, hogy a rajzolás kitölti az egész komponens területét és az átméretezésnél is mindig újrarajzolódik. Nézzük a Paint metódus kódjának első felét:

```
procedure TMd3Arrow.Paint;
var
  XCenter, YCenter: Integer;
begin
  {compute the center}
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;

  {draw the line and the arrow}
  case FDirection of
    adUp:
      with Canvas do
      begin
        MoveTo (XCenter, Height-1);
        LineTo (XCenter, YCenter);
        LineTo (Width-1, YCenter);
        LineTo (XCenter, 0);
        LineTo (0, YCenter);
        LineTo (XCenter, YCenter);
      end;
    adLeft:...
```

A teljes forrás megtalálható a CD Arrow1 könyvtárában a tesztprogrammal együtt.

A tesztprogram

Ahelyett, hogy telepítenénk a komponenst, írhatunk egy olyan tesztprogramot, amely futásidőben hozza létre a komponens egy példányát. A tesztprogram ATest névre hallgat, és ugyanabban az Arrow1 könyvtárban található, mint a komponenst definiáló egység.

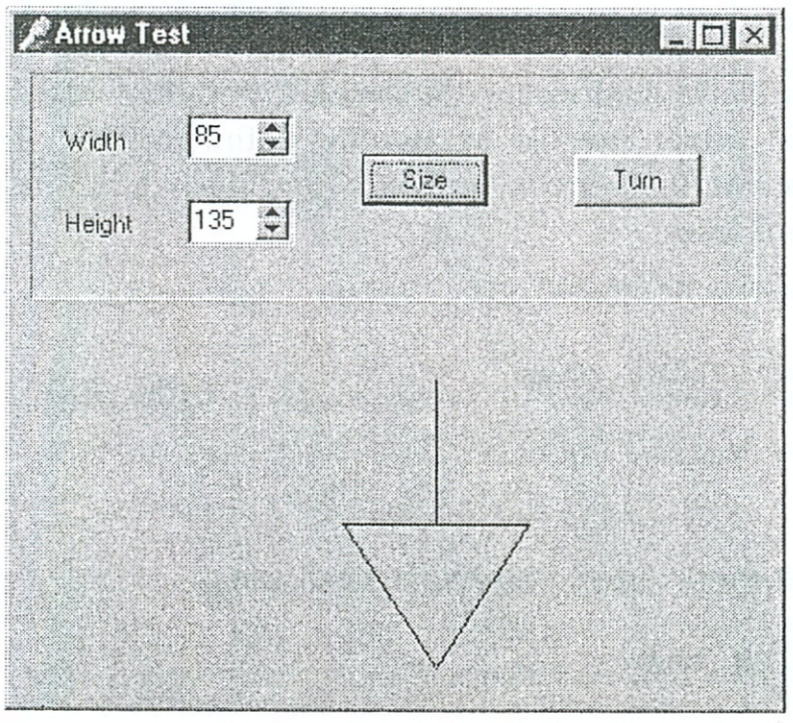
A form a komponensünkön kívül még tartalmaz egyéb elemeket is, amelyek az összetevők néhány tulajdonságának beállítására szolgálnak. Van két léptetőgombunk a nyíl szélességének és magasságának beállítására, egy gombunk a változtatások végrehajtására (Size) és egy másik gomb a nyíl irányának megváltoztatására (Turn), ez a Direction tulajdonságot változtatja meg. Ha a programot lefordítjuk, létrejön egy új nyíl komponens:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  A := TMd3Arrow.Create (self);
  A.Parent := self;
  A.Left := 150;
  A.Top := 150;
  A.Width := 20;
  A.Height := 30;
end;
```

A kódban az A egy egyszerű privát mezője a formnak, a típusa `TMd3Arrow`. A program létrehoz egy nyíl kontrolt és beállítja a Parent tulajdonságot magára a formra, hogy az a form belsejében jelenjen meg. A nyíl komponens akkor változik meg, ha a gombok valamelyikét lenyomjuk. A Size gomb a két SpinEdit komponens értékét használja fel a nyíl méreteinek állítására, a másik gomb, a Turn gomb pedig az irányt változtatja meg. Az aktuális értéket a Direction tulajdonságból olvassa ki, majd megnöveli egyel a Succ rendszerfüggvény hívásával. Ha elérte a maximális értéket, akkor a legkisebb érték lesz az új érték (a felsorolási típus szélsőértékeit a High és Low függvények hívásával tudhatjuk meg). Ezt a kódrészletet bármely más felsorolási típus vizsgálatára fel lehet használni. Végezetül álljon itt, az a kódrészlet, amely a Turn gomb OnClick eseményére fut le:

```
procedure TForm1.TurnButtonClick(Sender: TObject);
begin
  if A.Direction = High (TArrowDirection) then
    A.Direction := Low (TArrowDirection)
  else
    A.Direction := Succ (A.Direction);
end;
```

Mint ahogy az a 18.8. ábrán látható (a program futtatásával is kipróbálható), az új komponens működik. A probléma csak az, hogy a lehetőségek eléggé szűkösek. Nem változtathatjuk meg a vonalak vastagságát, a nyíl színét stb. Ezeket a problémákat a következő verzióban oldjuk meg.



18.8. ábra *Az Arrow1 komponens tesztelése*

Osztály típusú tulajdonságok hozzáadása a komponenshez

A nyíl megjelenését több tulajdonság írja le a komponensben. Az első két egyszerű tulajdonság az ArrowHeight és a Filled. A kódjuk most egész szokványos, de mindenestre álljon itt a forrás. A következőképpen definiáljuk a két új tulajdonságot:

```
type
  TMd3Arrow = class(TGraphicControl)
  private
    FArrowHeight: Integer;
    FFilled: Boolean;
    ...
    procedure SetArrowHeight (Value: Integer);
    procedure SetFilled (Value: Boolean);
  published
    property ArrowHeight: Integer read FArrowHeight
        write SetArrowHeight default 10;
    property Filled: Boolean read FFilled
        write SetFilled default False;
    ...
  end;
```

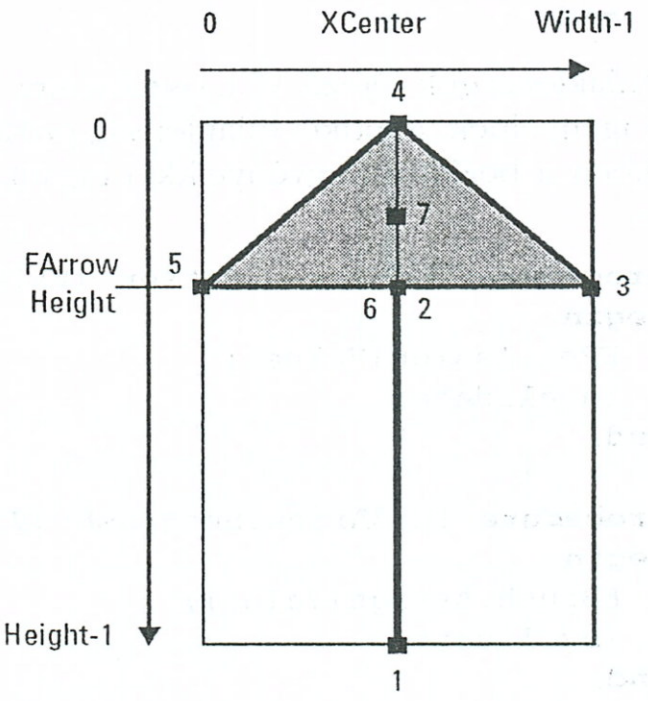
Az ArrowHeight tulajdonság határozza meg a nyíl hegyének méretét, a Filled tulajdonság pedig azt, hogy ki kell-e kitölteni. A Canvas aktuális ecsete határozza meg a használt színt, így aztán még egy tulajdonságot kell hozzáadnunk a komponenshez, hogy az ecsetet megváltoztassuk. Ha már ezt megtesszük, akkor miért ne definiálnánk egy másikat a toll számára is? Még elég sok tennivalónk van a komponens következő verziójának elkészültéig.

Először is, az ArrowHeight és Filled tulajdonságokat fogjuk kidolgozni. Mindkettőnek megvan a megfelelő privát mezője az osztály definíciójában, és ugyanígy megvannak azok a privát függvények, amelyek hozzáférnek az értékükhöz. A kód nagyon egyszerű lesz:

```
procedure TMd3Arrow.SetArrowHeight (Value: Integer);
begin
  if FArrowHeight <> Value then
  begin
    FArrowHeight := Value; Invalidate;
  end;
end;
procedure TMd3Arrow.SetFilled (Value: Boolean);
begin
  if FFilled <> Value then
  begin
    FFilled := Value; Invalidate;
  end;
end;
```

Legnagyobb hatással ez a két tulajdonság a Paint metódusra van, amely most már helyesen rajzolja meg a nyilat, működik a méretezés és a színezés. A Paint metódus új verziójában egy `case` szerkezet lesz, ebből az egyik ág a következő:

```
adUp:
  with Canvas do
  begin
    MoveTo (XCenter, Height-1);
    LineTo (XCenter, FArrowHeight);
    LineTo (Width-1, FArrowHeight);
    LineTo (XCenter, 0);
    LineTo (0, FArrowHeight);
    LineTo (XCenter, FArrowHeight);
    if FFilled then
      FloodFill (XCenter, FArrowHeight div 2,
        Pen.Color, fsBorder);
  end;
```



1) MoveTo (XCenter, Height-1);
2) LineTo (XCenter, FArrowHeight);
3) LineTo (Width -1, FArrowHeight);
4) LineTo (XCenter, 0);
5) LineTo (0, FArrowHeight);
6) LineTo (XCenter, FArrowHeight);
   if FFilled then
7) FloodFill (XCenter, FArrowHeight
   div 2, Pen.Color, fsBorder);

18.9. ábra *Egy új séma a felfelé mutató nyílhoz a Paint metódus új verziójának forrásában*

A kód hatását a 18.9. ábra érzékelteti, hasonlítsuk össze az előző verzióval a 18.7. ábrán. Egy terület kitöltéséhez a FloodFill eljáráshoz az első két paraméterben egy olyan pont koordinátáit kell megadnunk, amely a területen belül helyezkedik el. Mi most megközelítőleg a nyíl fejének középpontját fogjuk választani, de bármelyik pont jó lenne számunkra a nyíl fejét jelentő háromszögből.

A másik két tulajdonság osztály típusú (pontosabban `TPersistent` típusú, amelyeket a Delphi automatikusan kezel), azaz a komponensünknek kell gondoskodnia a megfelelő objektumok létrehozásáról, illetve megszüntetéséről. Mivel a felhasználók ezeknek a részobjektumoknak a tulajdonságait is állíthatják az Object Inspectorban, ezért az eseménykezelésüket komponensünkön belül kell megoldanunk. A két új tulajdonság definíciója:

```
type
  TMd3Arrow = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    ...
    procedure SetPen (Value: TPen);
    procedure SetBrush (Value: TBrush);
    procedure RepaintRequest (Sender: TObject);
  published
    property Pen: TPen read FPen write SetPen;
    property Brush: TBrush read FBrush write SetBrush;
  end;
```

Ezek a tulajdonságok létező VCL osztályokra hivatkoznak, amelyek speciális kezelést igényelnek. Mindkét tulajdonság privát mezőként deklarált a komponensünkben a beállító függvényeikkel együtt, amelyeket az alábbi egyszerű kód ír le:

```
procedure TMd3Arrow.SetPen (Value: TPen);
begin
  FPen.Assign(Value);
  Invalidate;
end;

procedure TMd3Arrow.SetBrush (Value: TBrush);
begin
  FBrush.Assign(Value);
  Invalidate;
end;
```

Figyeljük meg: az Assign metódust használjuk az egyszerű hozzárendelés helyett. Számos VCL osztálynak van saját Assign metódusa, amely az objektum adatainak frissítésére szolgál, de nem adhatók értékül nekik egy másik objektum.

A felhasználók gyakran változtatják majd a Pen vagy a Brush elemek tulajdonságait (például az Object Inspectort használva), ahelyett, hogy az összes értéket egyszerre lecserélnék. Hogyan tudjuk végrehajtani ezeket a részváltoztatásokat? A `TPen`, `TBrush` és más GDI osztályok egy speciális eseményt tartalmaznak erre a célra, ez az OnChange. Ha az OnChange eseményhez egy

metódust rendelünk ezeknél a komponenseknél, akkor ez a metódus minden egyes alkalommal lefut, ha az osztály valamely tulajdonsága megváltozik. Az Arrow komponensben ezt a kódrészletet a Create metódusba tesszük:

```
FPen.OnChange := RepaintRequest;
FBrush.OnChange := RepaintRequest;
```

A RepaintRequest kezelő a komponens private részében helyezkedik el, mint ahogy azt már az imént láttuk. A hatása, mint ahogy azt a neve is sugallja, a komponens újrafestése az Invalidate metódus hívásával:

```
procedure TMd3Arrow.RepaintRequest (Sender: TObject);
begin
  Invalidate;
end;
```

Most már használhatjuk a tollat és az ecsetet a rajzoláshoz, ehhez viszont újra meg kell változtatnunk a Paint metódust:

```
procedure TMd3Arrow.Paint;
var
  XCenter, YCenter: Integer;
begin
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;
  Canvas.Pen := FPen;
  Canvas.Brush := FBrush;
  case FDirection of
    adUp: with Canvas do
      ...
```

Az utolsó dolog, amire figyelnünk kell, hogy az FPen és az FBrush objektumok, és ezért ugyanúgy, mint bármely más objektumot, inicializálni kell őket. Ezért a Create konstruktort két újabb sorral kell bővítenünk. Itt a végleges kód:

```
constructor TMd3Arrow.Create (AOwner: TComponent);
begin
  {call the parent constructor}
  inherited Create (AOwner);
  {set the default values}
  FDirection := adRight;
  FArrowHeight := 10;
  FFilled := False;
  {create the pen and the brush}
  FPen := TPen.Create;
  FBrush := TBrush.Create;
  {set a handler for the OnChange event}
  FPen.OnChange := RepaintRequest;
  FBrush.OnChange := RepaintRequest;
end;
```

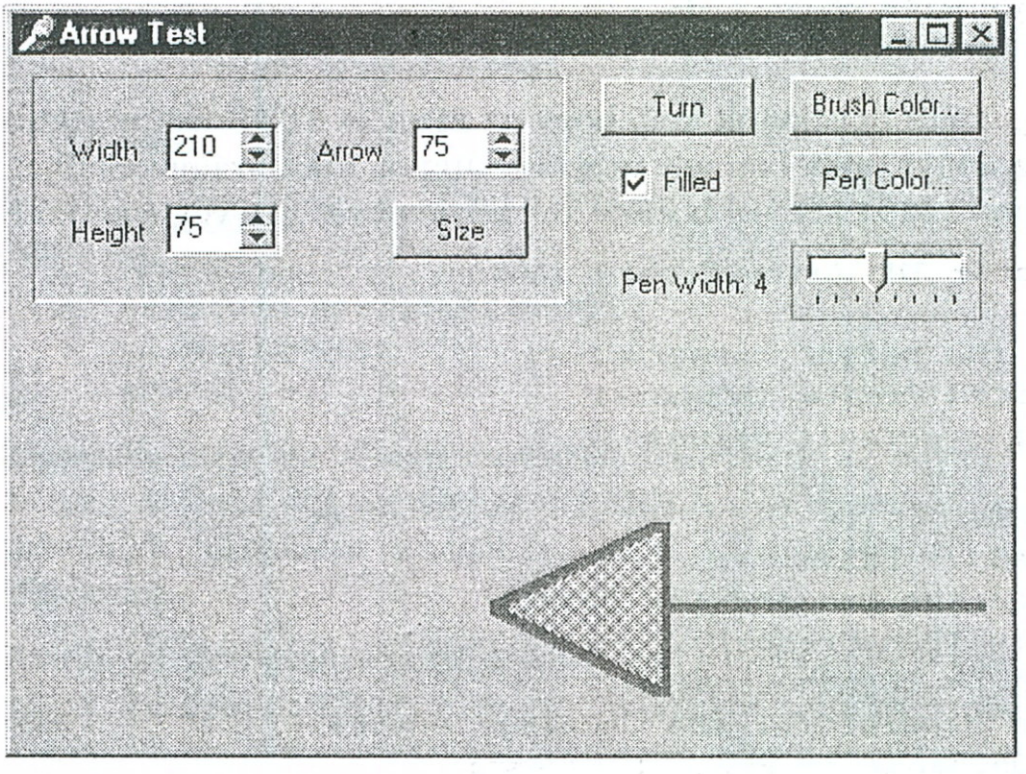
A komponensünkhöz tartozik egy destruktor is, melynek kódja a következő:

```
destructor TMd3Arrow.Destroy;
begin
  FPen.Free;
  FBrush.Free;
  inherited Destroy;
end;
```

Valószínűleg ez az első eset, hogy egy példában destruktort használunk (a destruktorokat a konstruktorokkal együtt az 5. fejezetben vezettük be). Most már láthatjuk, hogy igaz volt a fejezet elején tett megállapítás, miszerint a komponensírás az Object Pascal nyelv olyan elemeinek ismeretét feltételezi, amelyekre más esetekben ritkán van szükség.

A tesztprogram átírása

A komponens új verzióját sem telepítjük a Delphibe, ehelyett inkább átírjuk tesztprogramunkat. Ezt általános gyakorlatnak javasolom a komponensek készítésénél, amíg teszteljük őket, csak akkor telepítsünk, ha már úgy ítéljük meg, hogy a komponens teljesen elkészült (és remélhetőleg hibamentes). A tesztprogram új verziójában a formon néhány újabb komponens is van, további gombok, egy jelölőnégyzet, egy ColorDialog és egy Trackbar komponens, ahogy az a 18.10. ábrán látszik, amely a program futása közben készült.



18.10. ábra *Példa az Arrow2 komponens tesztprogramjának kimenetére*

A példaprogram kódja egyszerű továbbfejlesztése az előzőnek. A léptetőgomb értékét akkor használjuk fel, ha a felhasználó megnyomja a Size gombot, az ecset és a toll színének beállítása a ColorDialog komponens segítségével történik, a szokásos módon. A jelölőnégyzet beállítása a Filled tulajdonságra van hatással, a Trackbaron pedig a toll szélességét állíthatjuk be. Itt csak az utolsó kódrészletet mutatjuk be:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  PenLabel.Caption := 'Pen Width: ' +
    IntToStr (TrackBar1.Position);
  A.Pen.Width := TrackBar1.Position;
end;
```

A program fennmaradó része nagyon egyszerű, megtalálható a CD-n.

Saját esemény definiálása

Az Arrow komponens befejezéseként egy új eseményt készítünk. A legtöbb esetben az új komponensek a szülőosztály eseményeit használják. Például az Arrow komponens harmadik verziójában elérhetővé tettünk néhány eseményt egyszerű újradeklarálásukkal a `published` részben:

```
type
  TMd3Arrow = class(TGraphicControl)
  published
    property OnClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
```

Ennek a deklarációnak köszönhetően a fenti események (eredetileg a szülőosztályban deklaráltak) elérhetőek lesznek az Object Inspectorban, ha telepítjük a komponenst. A legtöbb komponens nagyon jól működik úgy is, ha csak egyszerűen felüldefiniáljuk a régi eseményeket, ezzel megváltoztatva eddigi viselkedésüket. Mindenestre hasznos és érdekes lesz látni, hogyan készíthetünk egy saját eseményt, bár ez a művelet legtöbbször nem szükséges.

Egy teljesen új esemény definiálásához az első lépésünk az, hogy az osztályhoz hozzáadjunk egy új típust, amely a létrehozandó esemény típusa lesz. Íme a `TArrow` osztály `private` részéhez hozzáadott kód:

```
FArrowDblClick: TNotifyEvent;
```

Ebben az esetben a `TNotifyEvent` típust használjuk, amely csak a Sender (küldő) paramétert kapja és egy általános eseménytípus a Delphiben, ezt

használják például az OnClick, vagy az OnDblclick eseményeknél is. A mező közelében definiálunk még egy nagyon egyszerű published tulajdonságot, amellyel közvetlen módon férhetünk hozzá az adathoz:

```
property OnArrowDblClick: TNotifyEvent
  read FArrowDblClick write FArrowDblClick;
```

Figyeljünk a konzekvens jelölési módra, az események neve az On szócskával kezdődik. Az FArrowDblClick metódus pointer aktiválódik (lefut a megfelelő függvény) az ArrowDblClick dinamikus metóduson belül. Ez azonban csak akkor történik meg, ha létezik megfelelő eseménykezelőnk:

```
procedure TArrow.ArrowDblClick;
begin
  if Assigned (FArrowDblClick) then
    FArrowDblClick (self);
end;
```

Ez a metódus a protected szekcióban deklarált a későbbi származtatás illetve változtatási lehetőségek miatt. Az ArrowDblClick metódust akkor hívja meg az eseménykezelő, ha a wm_LButtonDblClick Windows üzenetet kapja, de csak akkor, ha a duplakattintás a nyíl hegyén történt. Ennek a feltételnek a vizsgálatához a Windows API régiófigyelő függvényeit kell felhasználnunk.

A régió a képernyő egy behatárolt területe. Például megadhatunk egy sokszög régiót három ponttal, és ekkor egy háromszöget kapunk. A kitöltésnél adódó problémák elkerülése végett a pontokat egy TPoint típusú elemekből álló tömbben kell megadnunk, amelyben a pontok az óramutató járásával megegyező sorrendben helyezkednek el (erről a megközelítésről részletesebben a Windows API Súgóban olvashatunk a CreatePolygonalRgn leírásánál).

Ha definiáltuk a régiót, akkor a PtInRegion API hívással el tudjuk dönteni, hogy egy egéresemény azon belül, vagy kívül történt. A következő kódrészletben megtalálhatjuk a teljes eljárást a TMd3Arrow osztály új elemeinek típusdefiníciója után:

```
type
  TMd3Arrow = class(TGraphicControl)
  private
    FArrowDblClick: TNotifyEvent;
    procedure WMLButtonDlbClk(var Msg:TWMLButtonDblClk);
      message wm_LButtonDblClk;
    ...
  protected
    procedure ArrowDblClick; dynamic;
  published
    property OnArrowDblClick: TNotifyEvent
      read FArrowDblClick write FArrowDblClick;
  end;
```

```
procedure TMd3Arrow.WMLButtonDlbClk (
  var Msg: TWMLButtonDblClk);
var
  ArrowPoints: array [0..2] of TPoint;
  XCenter, YCenter: Integer;
  HRegion: HRgn;
begin
  {perform default handling}
  inherited;

  {compute the points}
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;
  case FDirection of
    adUp:
    begin
      ArrowPoints [0]:=Point (0, FArrowHeight);
      ArrowPoints [1]:=Point (XCenter, 0);
      ArrowPoints [2]:=Point (Width-1, FArrowHeight);
    end;
    adDown: ... // similar code
    adLeft:
    begin
      ArrowPoints [0]:=Point(FArrowHeight, Height - 1);
      ArrowPoints [1]:=Point (0, YCenter);
      ArrowPoints [2]:=Point (FArrowHeight, 0);
    end;
    adRight: ... // similar code
  end;

  {check whether the click took place
  in the arrow-point region}
  HRegion :=CreatePolygonRgn (ArrowPoints, 3, WINDING);
  if PtInRegion (HRegion, Msg.XPos, Msg.YPos) then
    ArrowDblClick;
  DeleteObject (HRegion);
end;
```

Az OnArrowDblClick esemény kipróbálása

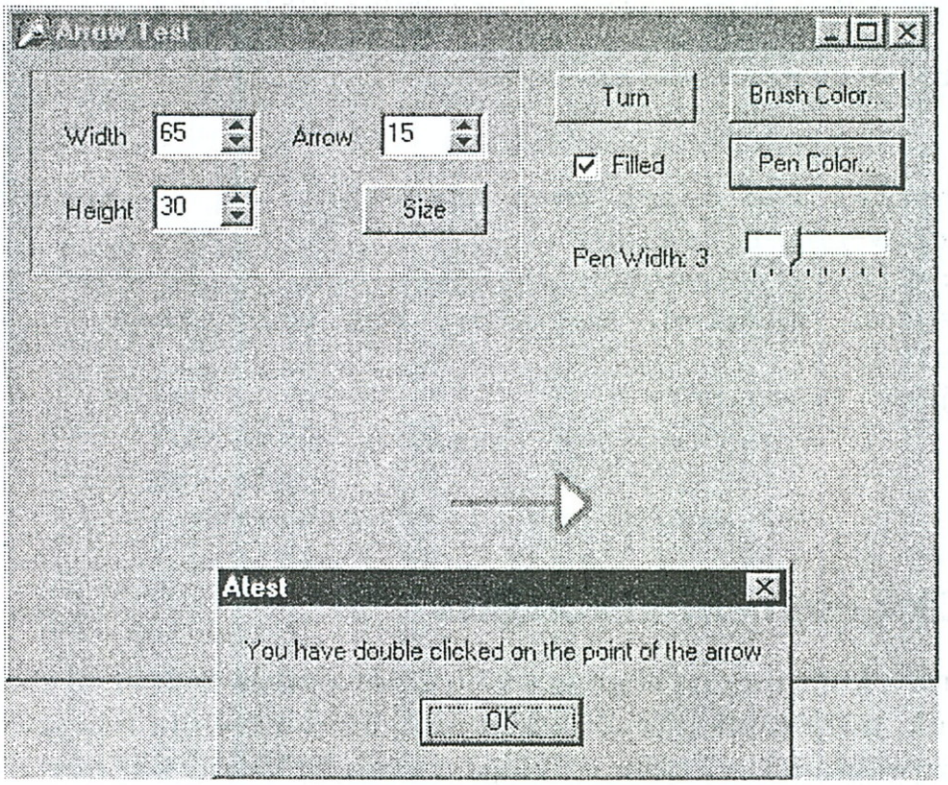
Most már elkészültünk a komponens végleges változatával, már csak az új eseményt kell tesztelnünk, mielőtt telepítenénk a komponenst a Delphi fejlesztői környezetébe. A tesztprogram harmadik verziója szintén az Arrow3 könyvtárban található a CD-n, és ugyanazt a formot használja, mint az előző verzió. A kódban természetesen megtalálható az új metódus, az ArrowDoubleClick. Az eljárás a form `public` részében található, a következőképpen néz ki:

```
procedure TForm1.ArrowDoubleClick (Sender: TObject);
begin
  ShowMessage ('You have double-clicked ' +
    'on the point of the arrow');
end;
```

Az eljárás akkor fut le, amikor az OnArrowDblClick esemény történik, ennek kiváltása a form Create metódusában történik:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  A := TMd3Arrow.Create (self);
  A.Parent := self;
  A.Left := 200;
  A.Top := 200;
  A.Width := 20;
  A.Height := 30;
  A.ArrowHeight := 15;
  A.OnArrowDblClick := ArrowDoubleClick;
end;
```

Mi lesz a kód eredménye? Ha futtatjuk a programot, és a nyíl háromszög alakú elejében duplán kattintunk az egérrel, akkor egy üzenet jelenik meg, ahogy az a 18.11. ábrán látszik. Ha az említett területen kívül kattintunk duplán az egér bal gombbal, akkor semmi nem történik.



18.11. ábra *Az üzenet megjelenik az Arrow3 komponens tesztprogramjában, ha a felhasználó kattint a nyílon*

Bitkép hozzáadása a komponenspalettához

Mielőtt telepítenénk elkészült új komponensünket, még egy lépést meg kell tennünk. Egy bitképet kell definiálnunk komponensünk számára. Az első két komponens, amelyeket a fejezet elején készítettünk, nem rendelkeztek saját bitképpel, ezek a szülőosztályuknál definiált bitképet örökölték meg. Ebben az esetben, amikor a szülőosztály nem rendelkezik saját bitképpel (absztrakt osztály, a komponenspalettán sem jelenik meg), a Delphi alapértelmezett bitképét kapják meg. Szerencsére, új bitkép létrehozása egyszerű feladat, ha ismerjük az ide vonatkozó szabályokat. Először is nyissuk meg az Image Editort, kezdjük egy új projektet, és válasszuk a DCR (Delphi Component Resource) projekttípust.

Tipp

A DCR fájlok egyszerű RES fájlok, más kiterjesztéssel. Létre lehet hozni őket bármilyen erőforrás-szerkesztővel, beleértve a Borland Resource Workshopot is, amely hatékonyabb eszköz, mint a Delphi Image Editor. Ha kész az erőforrás fájl, akkor egyszerűen csak meg kell változtatni a kiterjesztést RES-ről DCR-re.

Új bitkép hozzáadásához válasszuk a 24x24-es méretet. Most már készen állunk a rajzolásra. Egy másik fontos dolog az elnevezés: ebben az esetben nem konvenciókra kell figyelnünk, a Delphi határozza meg a nevet.

A bitkép nevének meg kell egyeznie a komponens nevével, az elején megtoldva egy T betűvel. Ebben az esetben a bitkép neve TMD3ARROW kell, hogy legyen. A név csak nagy betűkből állhat. Az olyan neveket, amelyekben vegyesen fordulnak elő kis és nagybetűk, a Delphi figyelmen kívül hagyja (például TMd3Arrow).

A DCR fájl neve meg kell, hogy egyezzen azzal az egységnévvel, amelyben a komponenst definiáltuk, tehát meg kell egyeznie a Pascal forrásfájl nevével (jelen esetben a fájl neve ARROW3.DCR lesz). A fájlnak ugyanabban a könyvtárban kell lennie, mint a DCU fájlnak.

Ha a komponens bitképe kész, akkor telepíthetjük a komponenst a Delphi fejlesztői környezetébe a Component/Install Component menüpont választásával, ahol egy csomagot kiválasztva (vagy egy újat létrehozva) a bitkép automatikusan beépül a csomag DLL-jébe. Ezután már használhatjuk az új komponenst a Delphi fejlesztői környezetében.

Az Arrow komponens utolsó próbája

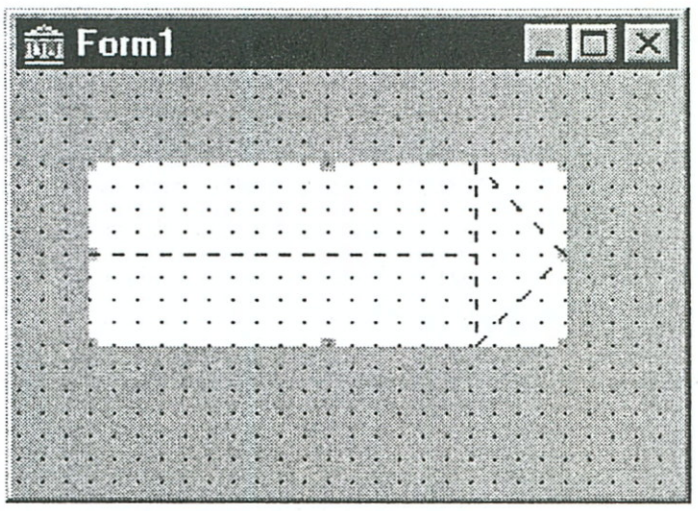
Ha telepítettük az Arrow komponens harmadik verzióját (amely még nincs az Md3Pack csomagban), akkor kipróbálhatunk egy egyszerű példát a teszteléséhez. Az telepítés után a komponens ugyanúgy használható, mint akármelyik másik, tulajdonságai beállíthatók az Object Inspectorban.

A tesztelésnek ez a formája nagyon fontos, gyakran előfordul, hogy a futásidőben létrehozott komponensekben néhány elem egészen másképpen viselkedik, mint ahogy azt várjuk. Tehát tesztelni kell minden lehetséges értékre a tulajdonságokat, hogy minden hibát megtalálhassunk, amelyeket esetleg az inicializálásnál, vagy máshol elkövettünk. Tesztelés közben én például két hibát találtam, amit kódolás közben követtem el.

Az első hiba az volt, hogy a komponensnek nem volt alapértelmezett mérete, így aztán amikor elhelyeztem a formra, a mérete egy pixel volt. A másik hiba akkor következett be, amikor váltogatni kezdtem a tollstílusokat. A legtöbb esetben a kitöltő algoritmus nem működött, ahogy az a 18.12. ábrán látható. Mivel ezek a problémák elég fontosak, hagytam az Arrow3 forrását, úgy ahogy van (hibákkal együtt) és a változtatásokat egy új verzióban (Arrow4) hajtottam végre, amelyet aztán telepítettem az Md3Pack csomagba.

Figyelem!

Fontos, hogy egyszerre csak egy verziót használhatunk egy komponensből, nem lehet két azonos nevű komponens (két osztály) a komponens könyvtárban.



18.12. ábra *Ha pontozott vonalat használunk tollnak az Arrow3-ban, problémák adódnak a kitöltésnél tervezési időben*

A hibák kijavítása az Arrow komponensben

Az első változtatás az Arrow komponens alapértelmezett méretének beállítása lesz, a Width és Height tulajdonság beállításával. Mindkét változtatást a Create metódusban és a tulajdonságok leírásánál is meg kell tenni:

```
property Width default 50;
property Height default 20;
```

Ez csak egyszerű újradeklarálása a szülőosztályban definiált tulajdonságoknak, ezúttal új alapértelmezett értékeket kapnak. Meg kell jegyeznem, hogy a szélességre és hosszúságra vonatkozó alapértelmezett érték megadása nem általános dolog a Delphi komponensekben, de én úgy gondolom, hogy célszerű őket megadni. A programozók általában nem változtatják meg a szokásos értékeket.

A többi változtatás a FloodFill (kitöltés) függvény hívásához kapcsolódik. Ha a FloodFill-t használjuk, akkor a régiót határoló vonalnak folyamatosnak kell lennie, nem lehet benne szakadás. A pontozott vonalakból soha nem lesz megfelelő határvonal, így aztán a FloodFill a területen kívül is mindent kitölt. Megérett tehát a helyzet arra, hogy újraírjuk a komponens Paint metódusát, felhasználva a TCanvas PolyLine és Polygon metódusait a FloodFill helyett. A PolyLine és a Polygon között az a különbség, hogy az előbbi egy vonalat rajzol, a második pedig ki is tölti a rajzolás során kapott sokszöget az aktuális ecsettel. Nem fogjuk észrevenni a változást a képernyőn, hacsak annyiban nem, hogy ez már működik, tehát jobb választás, mint a FloodFill.

Már láttuk, hogy lehet felépíteni egy sokszöget egy ponttömbből, amikor a nyílon kattintást teszteltük. Hasonló kódot fogunk most is használni. A probléma már csak az, hogy a PolyLine metódus nem zárja le automatikusan a sokszöget. Egy háromszög rajzolásához meg kell adni egy negyedik pontot is, amelynek egybe kell esnie az elsővel. Most pedig lássuk az Arrow komponens új Paint metódusának egy részét:

```
var
  ArrowPoints: array [0..3] of TPoint;
  begin
  ...
  case FDirection of
      adUp:
      begin
      Canvas.MoveTo (XCenter, Height-1);
      Canvas.LineTo (XCenter, FArrowHeight);
      ArrowPoints [0] := Point (0, FArrowHeight);
      ArrowPoints [1] := Point (XCenter, 0);
      ArrowPoints [2] := Point (Width-1, FArrowHeight);
      ArrowPoints [3] := Point (0, FArrowHeight);
    end;
```

```
    ...
  {draw the arrow point, eventually filling it}
  if FFilled then
    Canvas.Polygon (ArrowPoints)
  else
    Canvas.PolyLine (ArrowPoints);
end;
```

A TMd3Arrow osztály teljes definíciójának és összes metódusának forrását megtalálhatjuk az ARROW4.PAS egységben, az Arrow4 könyvtárban. A komponensnek ez a verziója található az Md3Pack csomagban.

Nyilak és alakzatok

Most már végre minden nagyszerűen működik, tesztelhetjük a komponensünket egy példaprojektben (az Arrow4 könyvtárban található a CD-n). A formon négy nyilat találunk, négy alakzatot, és egy időzítőt. A példaprogram célja, hogy a form területén egy színes formát mozgassunk. A színes elem feltételezhetően az eljárás aktív eleme lesz, amely az egyik négyzettől mozog a következő nyílig, aztán a következő négyzetig és így tovább. Hogy a mozgás fix időintervallumonként hajtódjon végre, egy időzítőt használunk (Timer komponens).

Amikor a form létrejön, a program eltárolja a négy alakzatot és a négy nyilat egy tömbben, amelynek az elemei grafikai komponensek, ahogy az itt látható:

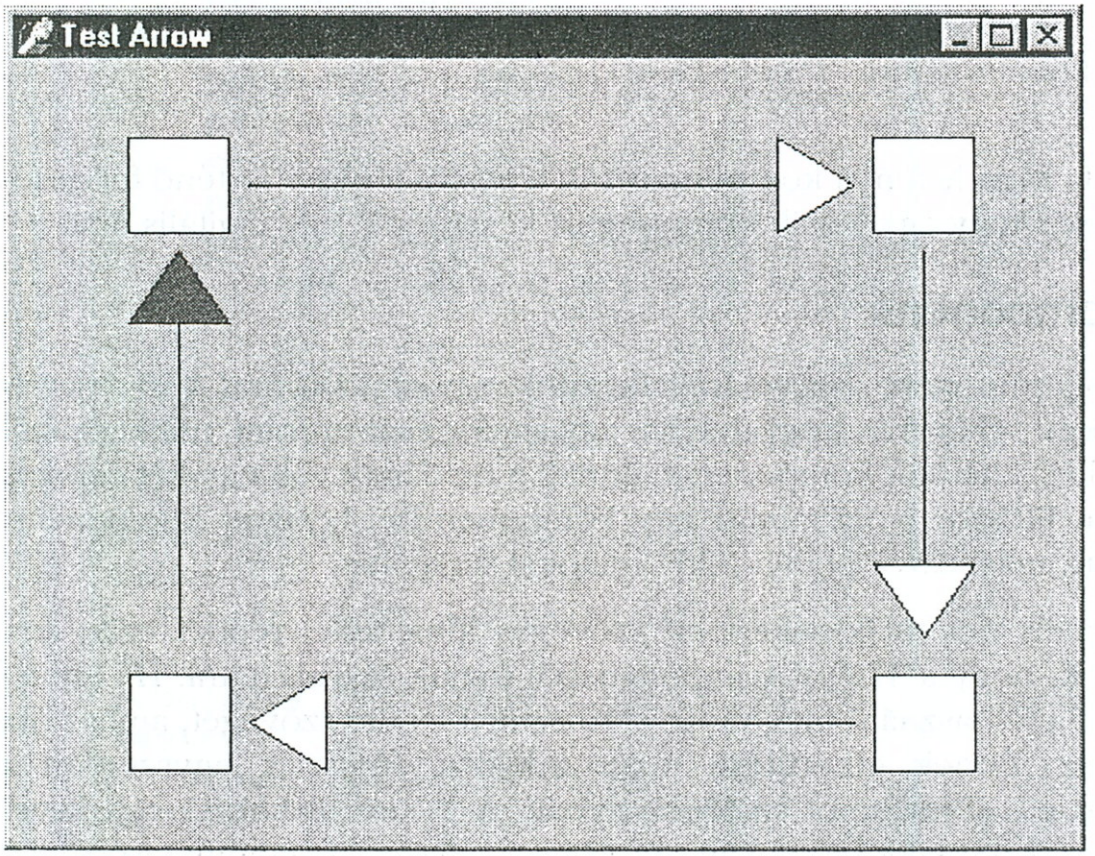
```
private
  Graph: array [1..8] of TGraphicControl;
```

A FormCreate metódus:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  Active := 1;
  Shape1.Brush.Color := clYellow;
  for I := 1 to 4 do
  begin
    Graph [I * 2 - 1] := (FindComponent (
      'Shape' + IntToStr (I)) as TGraphicControl);
    Graph [I * 2] := (FindComponent (
      'Md3Arrow' + IntToStr (I)) as TGraphicControl);
  end;
end;
```

Kicsit többet is mutat ez a példa, mint a komponensek egyszerű használata Delphiben. A FindComponent metódus egy gyermekkomponenst ad vissza, amennyiben az létezik. A metódus végignézi a form összes komponensét a

működése során. Ha a komponenseket tartalmazó tömb már fel van építve, akkor az aktív elem színe könnyen megváltoztatható, ahogy az a 18.13. ábrán látható.



18.13. ábra *A TArrow példa kimenete*

Ne felejtsük el visszaállítani annak az elemnek a színét, amely előzőleg aktív volt:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  // disables the active elements
  if Graph [Active] is TMd3Arrow then
    TMd3Arrow (Graph [Active]).Brush.Color := clWhite
  else
    TShape (Graph [Active]).Brush.Color := clWhite;
  // compute the new active element
  Active := Active mod 8 + 1;
  // enables the active element
  if Graph [Active] is TMd3Arrow then
    TMd3Arrow (Graph [Active]).Brush.Color := clRed
  else
    TShape (Graph [Active]).Brush.Color := clYellow;
end;
```

Ez minden? Igen, majdnem. Miért is ne jelenítsük meg a duplakattintásra az előbbi üzenet? A következő metódust kapcsoljuk mind a négy nyíl komponenshez az Object Inspector segítségével:

```
procedure TForm1.ArrowDoubleClick(Sender: TObject);
begin
  if (Sender as TMd3Arrow).Brush.Color = clRed then
    ShowMessage ('Double-click on active arrow');
end;
```

Ez a kis példa zárja le a nyíl komponensünk lépésről-lépésre történő fejlesztését. Itt az ideje, hogy egy másik komponenst készítsünk: egy digitális órát.

Az óra komponens

A következő komponens, amivel foglalkozunk, egy digitális óra. A 25. fejezetben látható egy példa arra, hogyan lehet egy analóg órát rajzolni, esetleg abból a kódból is készíthetünk komponenst. Ennek a példának van egy-két érdekes tulajdonsága. Először is, tartalmaz egy komponensbe ágyazott komponenst (egy időzítőt), másodszor pedig: „élő" adatokat szolgáltat.

Mivel a digitális óra tulajdonképpen szöveges kimenettel rendelkezik, arra gondolhatunk, hogy a TLabel komponensből fogjuk származtatni. Ha így teszünk, akkor a felhasználó meg tudja változtatni a címke szövegét, ami az óra kijelzője lesz. Ennek elkerülése miatt inkább válasszuk egyszerűen a `TCustomLabel` objektumot szülőosztálynak. A `TCustomLabel`-nek kevesebb `published` tulajdonsága van, egyébként teljesen ugyanúgy viselkedik, mint a `TLabel`. Más szavakkal, a `TCustomLabel`-nél el tudjuk dönteni, hogy mely tulajdonságok legyenek elérhetőek, és melyek maradjanak rejtve. A szülőosztálytól örökölt tulajdonságok újradeklarálása mellett a `TDigClock`-nak egy új tulajdonsága lesz, amelyet Active-nak nevezünk. Ez a tulajdonság azt fogja jelezni, hogy az óra jár-e. Mint az sejthető a fentiekből, az óra tartalmaz majd egy Timer komponenst. Az időzítőt nem tesszük publikus tulajdonsággá, mert el akarjuk kerülni, hogy a programozók közvetlenül hozzáférjenek. Ehelyett a Timer Enabled tulajdonságát fogjuk elfedni az óra Active tulajdonságával. A komponens teljes deklarációja:

```
type
  TMd3DigClock = class (TCustomLabel)
  private
    FTimer: TTimer;
    function GetActive: Boolean;
    procedure SetActive (Value: Boolean);
  protected
    procedure UpdateClock (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
```

```
  published
    property Align;
    property Alignment;
    property Color;
    property Font;
    property ParentColor;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property Transparent;
    property Visible;
    property Active: Boolean
      read GetActive write SetActive;
  end;
```

Szükségünk lesz két metódusra, amelyek az Active tulajdonságot írják és olvassák, mert ez nem a komponens adata tulajdonképpen, hanem a Timer komponensé:

```
function TMd3DigClock.GetActive: Boolean;
begin
  Result := FTimer.Enabled;
end;
procedure TMd3DigClock.SetActive (Value: Boolean);
begin
  FTimer.Enabled := Value;
end;
```

A Timer elérhetősége miatt felül kell definiálnunk az óra komponens konstruktorát. A Create metódus meghívja a szülőosztály megfelelő metódusát, majd létrehozza a Timer objektumot és beállítja az OnTimer eseménykezelőt:

```
constructor TMd3DigClock.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FTimer := TTimer.Create (self);
  FTimer.OnTimer := UpdateClock;
  FTimer.Enabled := True;
end;
```

Nincs szükség destruktorra, mert az FTimer objektum tulajdonosa a komponensünk, így aztán automatikusan felszabadul a helye a memóriában, ha a komponens megszűnik.

A komponens kódjának kulcseleme az UpdateClock eljárás:

```
procedure TDigClock.UpdateClock (Sender: TObject);
begin
  Caption := TimeToStr (Time);
end;
```

A metódus a Caption tulajdonságot használja, amely nem `published` tulajdonság, tehát a komponens felhasználója az Object Inspectorban nem fér hozzá. A kód eredményeképpen meg fog jelenni a pillanatnyi idő. Ez folyamatosan történik, mert a metódus a Timer komponens OnTimer eseményéhez van kötve.

Mint azt szoktuk, most is készítünk egy tesztprogramot a komponens kipróbálására. Ne felejtsük el telepíteni az Md3Pack csomagot a példaprogram megnyitása előtt, mert az tartalmazza a komponenst. A példaprogram a DigClock könyvtárban található a komponens teljes forrásával együtt.

Tömbtulajdonság definiálása

Most, hogy már egy sor érdekes komponenst készítettünk, nyilat és órát is, itt az ideje, hogy visszakanyarodjunk egy példához, amelyet még a fejezet elején vizsgáltunk, a tabulált listapanelhez. Most már teljessé tehetjük azt a példát is, tömb típusú tulajdonságot definiálva (most teszünk ilyet először). Semmi különlegesség nincs a tömbtulajdonságok definiálásában, de a technikájuk nem igazán dokumentált, ezért döntöttem úgy, hogy egy speciális példán keresztül bemutatom a használatát. Miközben ezt a példát kidolgozzuk, kapcsolatba kerülünk egy másik dologgal is, a párbeszédablak egységgel.

Az új komponens építésénél nem a régi forrást fogjuk átírni, hanem egy új komponenst készítünk TMd3TabbedList névvel. A különböző nevek miatt telepíthetjük egyszerre a gépünkre mind a két verziót, ugyanabban a csomagban, és ugyanabban a programban is használhatjuk őket, de ekkor különböző neveket kell adnunk a két egységnek.

Az új tabulált lista

A tabulált listapanel új verziója tartalmaz egy új tulajdonságot, amely a tabulátorpozíciókat állítja be. Ez egy tömbtulajdonság lesz, íme a definíciója:

```
property TabStops [Index: Integer]: Integer
  read GetTabStops write SetTabStops;
```

Ezt a tulajdonságot nem tehetjük az egység `published` részébe, mert a típusa nem osztálytípus és nem is beépített adattípus. Ez azt jelenti, hogy a felhasználó ezt a tulajdonságot csak futásidőben változtathatja meg, néhány sor kód írásával.

Megjegyzés

Ha `published`ként deklaráljuk ezt a tulajdonságot, akkor a fordító elfogadja, ami egy kicsit különösnek tűnik. Ha majd használni is akarjuk a komponenst Delphiben, akkor fogunk hibaüzenetet kapni.

Sokkal jobb megoldás lenne, ha a felhasználók a kezdeti értékeket közvetlenül az Object Inspectorban beírhatnák. Ha stringek listájával van dolgunk, akkor a kézenfekvő megközelítésnek a `TStringList` tulajdonság, illetve típus használata tűnik. Ha a lista elemei nem mind stringek, akkor a Delphi a `TCollection` típusát kell használnunk, amely csoportban tartalmaz `TCollectionItem` típusú elemeket, és minden elemre tárolja annak típusát. Ez a megoldás messze nem egyszerű. Ebben a példában én most egy könnyebb, nem szokványos megoldást választottam inkább (a Delphi fejlesztői kézikönyvben megtalálható a teljes példa Collection tulajdonság használatával).

Ahhoz, hogy megoldjuk azt a problémát, hogy fejlesztési időben is engedélyezzük a tulajdonság szerkesztését, egy új tulajdonságot kell felvennünk a tabulált listapanelünkhöz. Ez egy string lesz, benne az értékek lesznek felsorolva pontosvesszőkkel elválasztva. Mind a string, mind a tömb tulajdonság hozzáfér a privát mező értékeihez, amely egész számok tömbje lesz.

Megjegyzés

Nagyon fontos tudni, hogy a tulajdonságok adattípusa nem mindig egyezik meg az osztály által belül használt adattípusokkal. Az `interface` rész elválasztása az implementációtól az OOP lényegi eleme, és számos tulajdonság segíti ezt.

Íme a komponens osztályának teljes forráskódja:

```
type
  TTabsArray = array [0..9] of Integer;

  TMd3TabbedList = class(TListBox)
  private
    FTabStops: TTabsArray;
    function GetTabStops (Index: Integer): Integer;
    procedure SetTabStops (Index, Value: Integer);
    function GetTabsString: string;
    procedure SetTabsString (Value: string);
  protected
    procedure UpdateTabStops;
  public
    procedure CreateParams (
      var Params: TCreateParams); override;
    procedure CreateWnd; override;
    property TabStops [Index: Integer]: Integer
      read GetTabStops write SetTabStops;
```

```
  published
    property TabsString: string
      read GetTabsString write SetTabsString;
  end;
```

A felüldefiniált CreateParams metódus állítja be az lbs_UseTabStops listapanel stílust, hasonlóan a komponens előző verziójához. A felüldefiniált CreateWnd metódus állítja be a tömb kezdőértékeit, ezután az UpdateTabStops metódus hívásával frissíti a listapanel aktuális értékeit (ez egy saját metódus, amely a privát egész számos tömbből másolja az értékeket az aktuális Windows listapanelbe):

```
procedure TMd3TabbedList.CreateParams (
    var Params:TCreateParams);
begin
  inherited CreateParams (Params);
  Params.Style := Params.Style or lbs_UseTabStops;
end;
procedure TMd3TabbedList.CreateWnd;
var
  I: Integer;
begin
  inherited CreateWnd;
  for I := Low (FTabStops) to High (FTabStops) do
    FTabStops [I] := I * 100;
  UpdateTabStops;
end;
```

Ez a metódus a kezdeti tabulátorpozíciókat is beállítja. Most pedig lássuk azt a két metódust, amelyek hozzáférnek a tulajdonsághoz, a struktúrájuk nem lesz szokványos. Ennél a példánál mind a Get függvénynek, mind a Set eljárásnak eggyel több paramétere van: az index paraméter, amellyel a tulajdonság elemeihez férünk hozzá. A két metódus kódja nagyon egyszerű:

```
procedure TMd3TabbedList.SetTabStops (
  Index, Value: Integer);
begin
  if FTabStops [Index] <> Value then
  begin
    FTabStops [Index] := Value;
    UpdateTabStops;
    Invalidate;
  end;
end;
function TMd3TabbedList.GetTabStops (
  Index: Integer): Integer;
begin
  Result := FTabStops [Index];
end;
```

A SetTabStops metódus egyszerűen frissíti a belső tömböt, először meghívja az UpdateTabStops metódust, majd az Invalidate metódust a komponens újrarajzolásához. Két újabb metódusra lesz szükségünk, egy olyanra, amely az egész számokat stringgé konvertálja, és egy olyanra, amely a fordított konverziót végzi:

```
function TMd3TabbedList.GetTabsString: string;
var
  Text: string;
  I: Integer;
begin
  SetLength (Text, 100);
  Text := '';
  for I := Low (FTabStops) to High (FTabStops) do
    Text := Text + IntToStr (FTabStops ) + ';';
  Result := Text;
end;

procedure TTabList.SetTabsString (Value: string);
var
  Text: string;
  I, Len : Integer;
begin
  Text := Value;
  for I := Low (FTabStops) to High (FTabStops) do
  begin
    Len := Pos (Text, ';');
    FTabStops  := StrToIntDef (Copy (Text, 1, Len), 0);
    Delete (Text, 1, Len);
  end;
  UpdateTabStops;
  Invalidate;
end;
```

Az UpdateTabStops metódus a tulajdonképpeni kulcselem a komponensben. Beállítja a tabulátorpozíciókat a listapanel kontrolban, amelynek tulajdonosa a mi komponensünk. Ehhez a programnak egy `lb_SetTabStops` üzenetet kell küldenie a listapanelnek, paraméterként átadva a tabulátorpozíciók számát, valamint egy mutatót, amely az egész számok tömbjére mutat, amelyben a tabulátorpozíciók vannak tárolva:

```
SendMessage (Handle, lb_SetTabStops,
  1 + High (ConvertedTabs) - Low (ConvertedTabs),
  LongInt (@ConvertedTabs));
```

A párbeszédablak mértékegységeinek használata

A fenti kódban a ConvertedTabs egy `TTabsArray` típusú változó. Az FTabStops tömb nem használható közvetlenül, mivel a listapanel egy más mértékegységet használ. Igen, a listapanel tabulátorpozíciói párbeszédablak egységekben vannak, ha a listapanel egy általános ablakban van (bármely form ilyen). Nem akarom ráerőltetni senkire, hogy használja ezeket a mértékegységeket, dolgozhatunk pixelekben is. Ezért írtam a konverziós kódot az UpdateTabStops metódusba.

Először is, a párbeszédablak egységek alapja az ablak betűtípusa. Még precízebben: függ a képernyő felbontásától és az alapértelmezett betűmérettől. A párbeszédablak egységet úgy számolhatjuk ki, hogy vesszük az átlagos karakter szélességét és magasságát és osztjuk 4-el, illetve 8-al. Hogyan tudunk konverzióit készíteni párbeszédablak egységekre? A GetDialogBaseUnits Windows API függvény a párbeszédablak egységgel tér vissza (x az alsó két bájt, y pedig a felső kettő), így aztán már írhatjuk a következő kódot:

```
XDialog := XPixels * 4 div LoWord (GetDialogBaseUnits);
```

Sajnos ez a függvény nem megbízható. Csak olyan párbeszédablakokra (vagy listapanelekre) működik, amelyek a rendszerbetűtípust használják. Ezért inkább a kódban kell kiszámítanunk a betűtípus átlagos szélességét. Érdemesebb ezt a kódot használni, mint a Windows saját függvényét, amely a GetTextExtent API függvényre alapul. A Delphiben használhatjuk a `TCanvas` TextWidht metódusát. A Windows úgy számol, hogy veszi a 26 nagy és kisbetűt, kiszámolja a teljes szélességüket és osztja az összeget 52-vel:

```
HUnits := Canvas.TextWidth (
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz')
  div 52;
```

Természetesen először a betűtípust kell beállítani, hogy a következő listát láthassuk. Ha már megvan az alapegység, akkor könnyen kiszámíthatjuk az összes tabulátorpozíció új értékét a Windows listapanel kontroljában. Ennyi részletes elemzés után álljon itt az UpdateTabStops metódus teljes forrása:

```
procedure TMd3TabbedList.UpdateTabStops;
var
  I: Integer;
  HUnits: Integer;
  ConvertedTabs: TMd3TabsArray;
begin
  {determine the horizontal dialog box units used
  by the list box, which depend on its current font}
  Canvas.Font := Font;
  HUnits := Canvas.TextWidth (
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz')
    div 52;
```

```
    {convert the array of tab values}
    for I := Low (ConvertedTabs) to High (ConvertedTabs)
do
      ConvertedTabs[I] := (FTabStops [I] * 4) div HUnits;
    {activate the tabs stops in the list box,
    sending a Windows list box message}
    SendMessage (Handle, lb_SetTabStops,
      1 + High (ConvertedTabs) - Low (ConvertedTabs),
      LongInt (@ConvertedTabs));
end;
```

Most már telepíthetjük a komponenst a szokásos módon, gondoskodjunk új bitképről a listapanel bitképe helyett. Ez a komponens is (a bitképpel együtt) megtalálható az Md3Pack csomagban.

Tipp

Ha egy komponens egységét akarjuk lefordítani, akkor ugyanúgy mint máskor, meg kell nyitni a forrásállományt, mintha az egy projekt lenne (használjuk a Delphi Open Project menüpontját). Ha ez megtörtént, szintaktikus ellenőrzést végezhetünk, majd lefordíthatjuk DCU fájllá.

Fejléc és a tabulált listapanel

Most pedig kipróbáljuk a tabulált listapanelt egy tipikus példával: tegyünk fölé egy Header (fejléc) komponenst, hogy a felhasználó futásidőben tudja változtatni a tabulátorpozíciók helyét. A listapanel tabulátorpozíciói a program indulásakor kapnak értéket a FormCreate metódusban, ezután csak akkor változnak meg, ha a fejlécben az elválasztó elmozdul, ilyenkor lefut a Header1Sized metódus.

Mindkét eljárás egy egyszerű tömbön alapul, amely a tabulátorpozíciókat úgy számítja, hogy minden fejléchez az előző fejléc mérete adódik hozzá. Azaz, amíg a tabulátorpozíciók értékei abszolút értékek, addig a Header komponens elválasztói relatív értéket mutatnak az előző elválasztójelhez képest. A két metódus kódja:

```
for I := 0 to Header1.Sections.Count - 1 do
begin
  Md3TabbedList1.TabStops [I] :=
    Header1.SectionWidth [I] + Last;
  Last := Header1.SectionWidth ;
end;
```

A FormCreate metódus tartalmaz egy másik ciklust is, amely a többi tabulátorpozíció értékét feltölti egy kellően magas értékkel (a szám maga nem fontos, csak az a lényeg, hogy ne legyen probléma belőle a későbbiekben):

```
for I := Header1.Sections.Count to 9 do
    Md3TabbedList1.TabStops [I] := 1000;
```

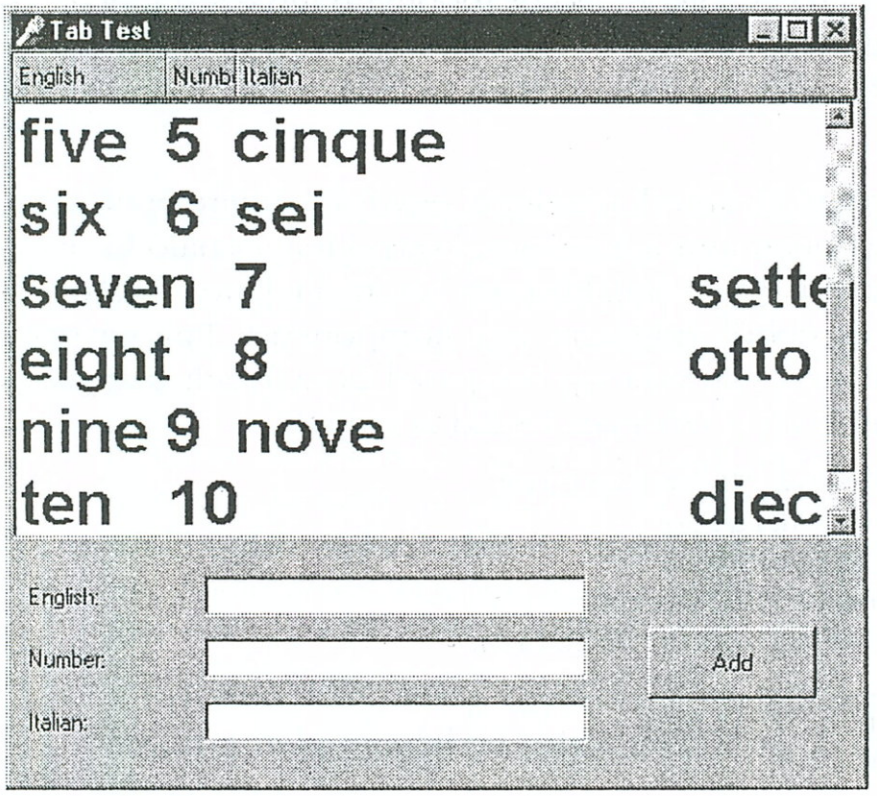
Ha lefuttatjuk a programot, akkor a listapanel tabulátorpozíciók automatikusan átállítódnak a fejlécpozíciókkal szinkronban, így jó ránézni a képernyőre. Amint az a 18.14. ábrán látszik, a dolog működik bármilyen karakterkészlettel, nem csak a rendszerfonttal. A kritikus kód nem ebben a példában van, de a tabulált listapanel használja, és éppen ez a Delphi komponensek erőssége. Megjegyezzük, hogy a későbbiekben továbbfejlesztjük ezt a példát, készítünk egy egyszerű komponenst, amely rendelkezik egy fejléccel is. Ez egy valóban hasznos komponens lesz, mert a Microsoft a Windows 95-höz nem készített ilyen speciális ListView kontrolt, amelybe ez a tulajdonság be lett volna építve.

18.14. ábra *A TabTest2 példaprogram*

Másrészről vegyük észre, hogy a listapanel tabulátorkarakterei valóban tabulátorok. Ez azt jelenti, hogy ha a második oszlopot túl közel mozgatjuk az elsőhöz, akkor néhány karakter nem jelenik meg, az első oszlop szavainak számától függően. Erre a viselkedésre láthatunk példát a 18.15. ábrán. Ez nem hiba,

a tervezés maga adja ezt a jelenséget. Hogy ezt a problémát kiküszöböljük, ki kell számítani a tabulátor méretét, ellenőrizve minden string hosszát az első oszlopban az aktuális betűtípussal.

Jó ez a komponens, de hiányzik belőle egy fontos tulajdonság: habár be tudjuk állítani a tabulátorok kezdőértékeit tervezési időben is, felhasználva a pontosvesszőkkel tagolt stringünket, azért sokkal jobb lenne (és könnyebben kezelhetőbb) ha egy szerkesztőt tudnánk biztosítani ennek a tulajdonságnak. Ennek megoldására egy speciális tulajdonságszerkesztőt kell írnunk, valami olyasmit, amiről a következő fejezetben lesz szó.



18.15. ábra *A tabulált listapanel viselkedése, ha a tabulátorpozíciók túl közel vannak*

Egy nem vizuális komponens készítése

Az utolsó komponens, amelyet ebben a fejezetben megvizsgálunk, teljes mértékben elüt az eddig bemutatott komponensektől. Az ablakalapú komponensek és egyszerű grafikai komponensek után most egy nem vizuális (képernyőn meg nem jelenő) komponenst készítünk.

Az alapötlet az, hogy maguk a formok is komponensek. Ha egy különlegesen hasznos formot készítünk, amely számos projektben hasznunkra lehet, akkor két lehetőségünk van: hozzáadhatjuk az Object Repository-hoz, vagy

készíthetünk belőle egy komponenst. A második megközelítése a problémának sokkal bonyolultabb, mint az első, de könnyebben használhatjuk az új formot, és lehetőségünk nyílik a form terjesztésére a forráskód továbbadása nélkül. Például megpróbálhatunk készíteni olyan párbeszédablakokat, amelyek a Delphi beépített párbeszédablakaihoz hasonlóan működnek.

Párbeszédablak a komponensben

Ahhoz, hogy egy párbeszédablakot fel tudjunk használni egy komponensben, első lépésként párbeszédablak kódját kell megírnunk, a Delphi példáját követve. Vegyünk elő egy új formot és dolgozzunk rajta a szokásos módon. Ha egy komponens egy formon helyezkedik el, akkor többnyire tervezési időben vizuálisan tervezhetjük meg a kinézetét. Természetesen, ha egyszer a párbeszédablakot elkészítettük, akkor a későbbiekben nem vizuális módon tudjuk definiálni.

A párbeszédablak, amelyet el kívánunk készíteni, egy listapanelre épül, azért, mert ez alkalmazható legjobban arra a feladatra, hogy a felhasználó kiválasszon egy stringet egy listából. Kicsit átalakítottam ezt az általános viselkedést egy párbeszédablakban és ebből építettem egy komponenst. Egy egyszerű formról van szó (ListBoxForm), van rajta egy listapanel és az ilyenkor szokásos OK és Cancel gombok, mint azt a szöveges leírása is mutatja.

```
object ListBoxForm: TListBoxForm
  BorderStyle = bsDialog
  Caption = 'ListBoxForm'
  object ListBox1: TListBox
    OnDblClick = ListBox1DblClick
  end
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

Párbeszédablak formunk egyetlen metódusa a duplakattintás eseményt használja, ilyenkor a párbeszédablak bezáródik, és a hatás ugyanaz, mintha a felhasználó az OK gombot nyomta volna meg:

```
procedure TListBoxForm.ListBox1DblClick(Sender:
    TObject);
begin
  ModalResult := mrOk;
end;
```

Ha a form már működik, elkezdhetjük variálni a forráskódot, újradefiniálhatjuk a komponenst, kivethetjük a form deklarálását a globális változók köréből, átmozgatva a form típusdefinícióját az implementációs részbe. A műveletek eredménye megtekinthető a példa forráskódjában.

Megjegyzés

Formalapú komponensek esetén használhatjuk a Pascal forráskód fájlokat, egyet a formnak, egyet pedig a komponensnek. Ebben a példában ehelyett inkább a form osztályt és a komponensosztályt is egy egységbe tettük. A form osztály implementációs részben való deklarálásával egyben elrejtettük a komponens felhasználói elől is.

A legfontosabb tulajdonsága ezeknek a műveleteknek, hogy a TMd3ListBox komponens nem vizuális komponens lesz. Ezt az határozza meg, hogy a közvetlen szülőosztálya a `TComponent` osztály. A komponensnek három `published` és egy `public` tulajdonsága van. A három `published` tulajdonság:

- A Lines tulajdonság egy `TStrings` típusú objektum, amelyhez két metódus tartozik, a GetLines és a SetLines. A második metódus az Assign eljárást használja az új értékek bemásolására a tulajdonságnak megfelelő privát mezőbe. Ez a belső objektum a Create konstruktorban jön létre, és a Destroy metódusban szűnik meg létezni.
- A Selected tulajdonság egy integer típusú mező, amely közvetlenül fér hozzá a neki megfelelő privát mezőhöz. A listából kiválasztott elemet tárolja.
- A Title tulajdonság a párbeszédablak fejlécének megváltoztatására szolgál.

A publikus tulajdonság a SelItem, amely csak olvasható, automatikusan visszaadja a listában éppen kiválasztott elemet. Fontos, hogy ez a tulajdonság nem tárol semmilyen adatot, egyszerűen csak más tulajdonságokra hivatkozik, tulajdonképpen csak az adat virtuális reprezentációjának tekinthető.

```
type
  TMd3ListBoxDialog = class (TComponent)
  private
    FLines: TStrings;
    FSelected: Integer;
    FTitle: string;
    function GetSelItem: string;
    procedure SetLines (Value: TStrings);
    function GetLines: TStrings;
```

```
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function Execute: Boolean;
    property SelItem: string read GetSelItem;
  published
    property Lines: TStrings read GetLines
        write SetLines;
    property Selected: Integer read FSelected
        write FSelected;
    property Title: string read FTitle write FTitle;
  end;
```

A kód legnagyobb része ebben a példában az Execute metódusban található, ez egy függvény, amely igaz vagy hamis értékkel tér vissza attól függően, hogy a párbeszédablak eredménye mi volt (mrOk vagy mrCancel). Ez a működés teljesen analóg a legtöbb beépített Delphi párbeszédablak komponensének Execute metódusának működésével. Az Execute metódus dinamikusan létrehozza a formot, beállítja a komponens által használt tulajdonságok kezdőértékeit, megjeleníti a párbeszédablakot, majd ha az eredmény korrekt, visszatér és frissít:

```
function TMd3ListBoxDialog.Execute: Boolean;
var
  ListBoxForm: TListBoxForm;
begin
  if FLines.Count = 0 then
    raise EStringListError.Create (
        'No items in the list');
  ListBoxForm := TListBoxForm.Create (self);
  try
    ListBoxForm.ListBox1.Items := FLines;
    ListBoxForm.ListBox1.ItemIndex := FSelected;
    ListBoxForm.Caption := FTitle;
    if ListBoxForm.ShowModal = mrOk then
    begin
      Result := True;
      Selected := ListBoxForm.ListBox1.ItemIndex;
    end
    else
      Result := False;
  finally
    ListBoxForm.Destroy;
  end;
end;
```

Figyeljük meg, hogy a kód tartalmaz egy `try-finally` blokkot arra az esetre, hogyha a párbeszédablak megjelenítése közben valamilyen hiba lépne fel, akkor a form azonnal megszűnik. Használjuk a kivételt arra az esetre is, ha a

lista üres, amikor a felhasználó használni akarja a párbeszédablakot. Ez tervezési fázisban előjövő hiba, így egy kivételkezelő rutin megfelel az elhárítására. A komponens további metódusai nagyon egyszerűek lesznek:

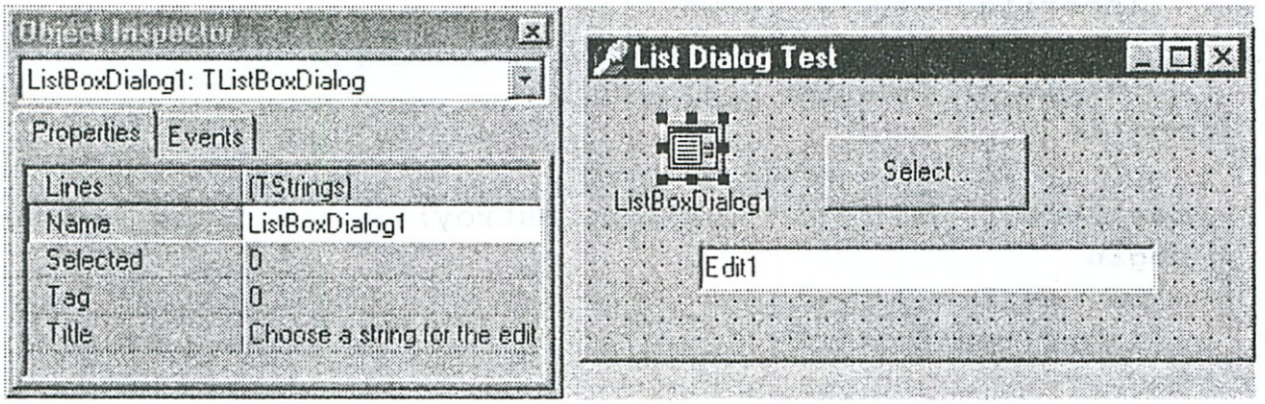
```
constructor TMd3ListBoxDialog.Create(AOwner:TComponent);
begin
  inherited Create (AOwner);
  FLines := TStringList.Create;
  FTitle := 'Choose a string';
end;

destructor TMd3ListBoxDialog.Destroy;
begin
  FLines.Free;
  inherited Destroy;
end;

function TMd3ListBoxDialog.GetSelItem: string;
begin
  if Selected >= 0 then
    Result := FLines [Selected]
  else
    Result := '';
end;

function TMd3ListBoxDialog.GetLines: TStrings;
begin
  Result := FLines;
end;

procedure TMd3ListBoxDialog.SetLines (Value: TStrings);
begin
  FLines.Assign (Value);
end;
```

Természetesen, mivel kézzel írjuk a komponens kódját, és az eredeti form forráskódjába illesztjük, ezért nem szabad megfeledkeznünk a Register eljárásról. Ez egy a számos változtatás közül, amelyeket a szokásos formstruktúrában elkövettem a form kódjának írása közben, mint ahogy az a teljes forráslistában is látható (megtalálható a CD-n a ListDial könyvtárban).

Nem vizuális komponensek használata

A komponens most már kész, már csak egy bitképről kell gondoskodnunk. A nem vizuális komponenseknél a bitképek nagyon fontosak, mert különben nem tudjuk őket feltenni a komponenspalettára (nem tud örökölni a szülőosztálytól bitképet, mert a `TComponent`-nek nincs), így aztán elég körülményes lenne elhelyezni őket egy formra. Hát akkor készítsünk egy bitképet, majd

telepítsük a komponenst, utána pedig készítsünk egy egyszerű projektet a teszteléshez. A tesztprogram formja mindössze egy gombot, egy szerkesztőmezőt és egy példányt tartalmaz új nem vizuális komponensünkből, mint ahogy az a 18.16. ábrán jól látható.



18.16. ábra *Az LdTest példa formja, az új, nem vizuális komponenssel*

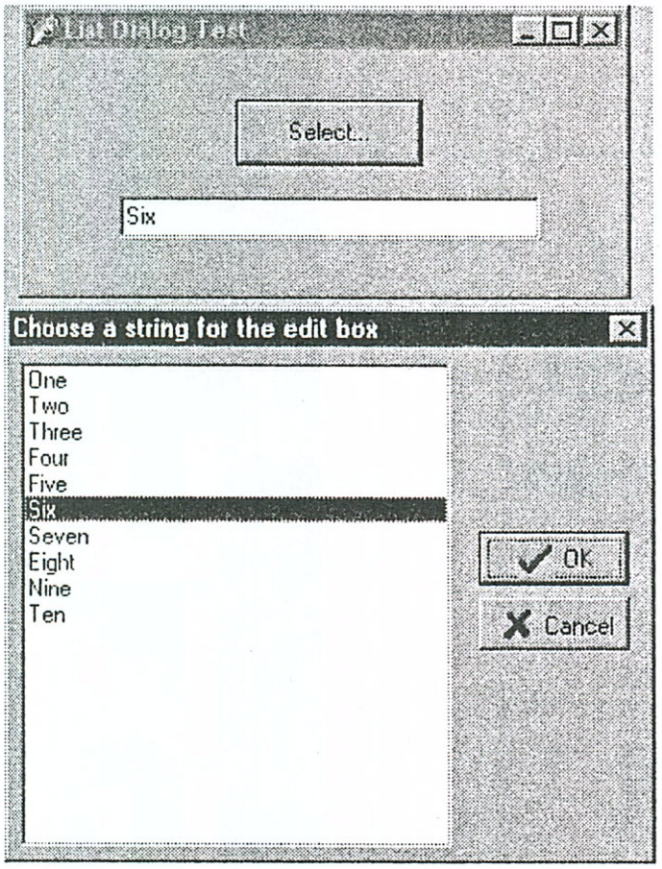
Írjunk pár sor kódot, mégpedig a gomb OnClick eseményére:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Md3ListBoxDialog1 do
    if Execute then
      Edit1.Text := SelItem;
end;
```

Ez minden, ami a formra helyezett párbeszédablak futtatásához kell, mint az a 18.17. ábrán látszik. Úgy gondolom, hogy az ilyen típusú fejlesztés érdekes megközelítése általános célú párbeszédablakok készítésének.

Hogyan tovább?

Ezt a fejezetet Delphi komponensek tervezésének és készítésének szenteltük. Láttuk, hogyan kell különböző típusú tulajdonságokat definiálni, hogyan adhatunk hozzájuk eseményeket, hogyan tudunk definiálni és felülírni komponensmetódusokat. Különböző példákat láttunk komponensekre, láttunk olyat, amely csak egyszerű változtatásokat végzett egy már meglévő komponensen, láttunk új grafikus komponenst, és a fejezet megkoronázásaképpen láttunk egy párbeszédablakot egy komponensbe ágyazva. A komponensek építése közben a Windows programozás újabb területeire jutottunk el. Általánosságban elmondható, nem szokatlan dolog a Windows API függvényeinek közvetlen használata Delphi komponensek készítése közben.



18.17. ábra *Az LdTest példa megmutatja a ListDial komponensbe ágyazott párbeszédablakunkat*

A komponenskészítés nagyon ügyes technika szoftvereink újrahasznosítása érdekében, de készítésük közben törekedjünk arra, hogy minél jobban illeszkedjenek a Delphi fejlesztői környezetébe. Hogy ezt elérjük, célszerű tulajdonság és komponensszerkesztőket készítenünk, erről a következő fejezetben lesz szó. A 19. fejezet megkísérel egy általános bemutatást adni a Delphi ToolsAPI eszközeiről, amelyek segítségével a programozónak lehetősége nyílik új varázslók és verziókövető rendszerek készítésére.

19. FEJEZET

A komponensek és a ToolsAPI

E fejezet tartalma:

- Tulajdonság-szerkesztők (Property Editor) készítése
- Komponens-szerkesztő készítése
- Egyszerű varázslók készítése
- A ToolsAPI további lehetőségei
- Tulajdonságok elérése név szerint

A komponensek és a ToolsAPI

Komponensek készítésével átalakíthatjuk a Delphi környezetet, ezáltal gyorsabban készíthetjük el programjainkat anélkül, hogy az alacsonyabb szintű technikákról alaposabb ismeretekkel rendelkeznénk. A Delphi környezet ugyancsak támogatja a külső cégek által készített kiegészítések beépítését rendszerünkbe. Kiemelkedően fontos, hogy egyszerűen beágyazhatunk az Object Inspectorba saját magunk által készített tulajdonság-szerkesztőket, és komponens-szerkesztőkkel bővíthetjük a formtervezőt. Egy további lehetőség a varázslók készítése, amihez általában hozzátartozik az is, hogy új menüpontokat építhetünk be a Delphi menübe. Ezen technikák mellett további beépített interfészek is rendelkezésre állnak azoknak a fejlesztőknek, akik új eszközöket akarnak készíteni. Ezeknek a módszereknek a használatához azonban a Delphi működésének alaposabb ismerete szükséges, és tisztán kell látni számos más technikát is.

Ezekből az okokból kifolyólag ebben a fejezetben csak néhány alapmódszerét mutatjuk be ennek a témának, a teljesség igénye nélkül. A Delphi környezet kiegészítési lehetőségeiről részletesebb információkat az ide vonatkozó szakirodalomból, például a Delphi Fejlesztői kézikönyvből tudhatunk meg.

Tulajdonság-szerkesztők (property editor) készítése

Az előző fejezetben egy olyan címkézett listapanel komponenst írtunk, melyben egy speciális karakterlánc tulajdonságban tároltunk egész számokat a címkék pozíciójának megfelelően. Ez az eljárás lehetővé teszi, hogy a felhasználó az Object Inspector segítségével már tervezéskor beállítsa a megfelelő értékeket. Azonban nem nevezhetjük kényelmesnek, hogy a részstringeket az Object Inspector egyetlen sorában kell megadnunk, pontosvesszőkkel elválasztva. Ehelyett létrehozhatunk egy saját tulajdonság-szerkesztőt.

A tulajdonság-szerkesztők típusai

Mindegyik tulajdonság-szerkesztőnek az absztrakt `TPropertyEditor` osztály leszármazottjának kell lennie. Ez az osztály a DsgnIntf (Design Interface, tervezői felület) rendszeregységben került definiálásra. A Delphiben azonban már definiáltak néhány speciális tulajdonság-szerkesztőt a karakterláncokhoz (`TStringProperty` osztály), az egészekhez (`TIntegerProperty` osztály), a karakterekhez (`TCharProperty` osztály), a felsorolt típusokhoz (`TEnumProperty` osztály), a halmazokhoz (`TSetProperty` osztály) és még sok-sok további szerkesztőt színekhez, betűtípusokhoz, rajzoló-tollakhoz, stringlistákhoz és még sok minden máshoz.

A saját tulajdonság-szerkesztőkben a GetAttributes függvényt felül kell definiálnunk úgy, hogy a szerkesztőről bizonyos értékek visszaadásával információkat adjon. A beállítható jellemzők közül a következő táblázat mutat be néhányat.

Tulajdonság	Magyarázat
`paValueList`	A tulajdonság-szerkesztő értékek listáját jeleníti meg az Object Inspector egy kombinált listájában. A tulajdonság-szerkesztőnek a GetValues metódust kell felülírnia. Ez a függvény fogja megadni a Delphinek a listában szereplő értékeket.
`paSubProperties`	A tulajdonság-szerkesztő néhány altulajdonságot kér be, amikor a felhasználó a tulajdonság neve mellett lévő pluszjelre kattint. Ennek a tulajdonság-szerkesztőnek a GetProperties metódus felülírásával kell megadnia az altulajdonságok listáját.
`paDialog`	A tulajdonság-szerkesztő egy saját szerkesztőt jelenít meg egy különálló párbeszédablakban, amely akkor jelenik meg, amikor a felhasználó az Object Inspector mezőjében található nyomógombon kattint. A tulajdonság-szerkesztőnek az Edit metódust kell felülírnia, és ennek meghívásakor kell megjelenítenie a párbeszédablakot.
`paMultiSelect`	Azt jelzi, hogy ez a tulajdonság akkor is megjeleníthető az Object Inspectorban, ha a felhasználó egynél több, azonos típusú komponenst egyszerre jelöl ki, vagy különböző típusúakat, de olyanokat, melyekben egyaránt szerepel ez a tulajdonság. Ez a kapcsoló általában engedélyezett.
`paAutoUpdate`	Azt jelzi, hogy a tulajdonság értékét a rendszer folyamatosan frissíti, miközben a felhasználó változtatja az értéket a szerkesztőben (például miközben a Caption tulajdonsághoz új értéket gépelünk be, az folyamatosan látható a képernyőn is). Ha ez az érték nincs beállítva, akkor az csak akkor lesz frissítve, amikor a szerkesztés véget ért.
`paSortList`	Az Object Inspector által megjelenített listában az elemek rendezve szerepelnek (e beállításnak a paValueList tulajdonsággal együtt van értelme).
`paReadOnly`	Az Object Inspector nem engedélyezi a tulajdonság megváltoztatását a normális, beépített szerkesztőben (ez az a terület, ami az Object Inspector ablak jobb oldalán található). Mindemellett egy tulajdonság-szerkesztővel megváltoztatható az érték. Gondoljunk arra, hogy ez már nem vonatkozik a csak olvasható tulajdonságokra, amelyek az Object Inspectorral nem változtathatóak meg. Példánkban ez az, amire nekünk is szükségünk van, hiszen tervezéskor a stringet listában akarjuk megjeleníteni, de nem akarjuk, hogy a felhasználó közvetlenül az Object Inspectorból megváltoztathassa az értékeket.

Szerkesztő a TabsString tulajdonsághoz

A TabsString tulajdonság esetén egy párbeszédablakot megjelenítő egyéni tulajdonság-szerkesztőt szeretnék megvalósítani. Mivel a tulajdonság string típusú, így az új szerkesztőt érdemes a karakterláncok tulajdonság-szerkesztőjéből származtatni. Ez látható az új tulajdonság-szerkesztő osztályunkban:

```
uses
  DsgnIntf;
type
  TMd3IntListProperty = class (TStringProperty)
  public
    function GetAttributes:TPropertyAttributes;
        override;
    procedure Edit; override;
end;
```

Tipp

A Borland konvenciói szerint a tulajdonság-szerkesztő osztályok neve mindig `Property` végződésű, a komponens-szerkesztőké pedig `Editor`.

A GetAttributes függvénnyel azt akarjuk tudatni a Delphivel, hogy egyéni szerkesztőnkben egy párbeszédablakot szeretnénk megjeleníteni, és megtiltjuk a string közvetlen szerkesztését az Object Inspectorban:

```
function TMd3IntListProperty.GetAttributes:
  TPropertyAttributes;
begin
  Result := [paDialog, paReadOnly];
end;
```

Nem engedhetjük meg, hogy a felhasználó egyszerre több komponens tulajdonságát beállítsa, mert ez az Edit metódust túlságosan elbonyolítaná. Ebben az eljárásban, a komponens címkéit tartalmazó tömböt közvetlenül szeretnénk elérni, anélkül, hogy felhasználnánk a TabsString tulajdonságot (amit a szerkesztő változtatni fog). Íme az Edit metódus kódja:

```
procedure TMd3IntListProperty.Edit;
var
  PEForm: TIntListPEForm;
  Tabs: TMd3TabbedList;
  I: Integer;
```

```
begin
  PEForm := TIntListPEForm.Create (Application);
  try
    Tabs := GetComponent (0) as TMd3TabbedList;
    for I := Low (TTabsArray) to
        High (TTabsArray) do
      PEForm.Memo1.Lines.Add (
        IntToStr (Tabs.TabStops [I]));
    if PEForm.ShowModal = mrOK then
    begin
      for I := Low (TTabsArray) to
          High (TTabsArray) do
        Tabs.TabStops [I] := StrToIntDef (
          PEForm.Memo1.Lines [I], 0);
      // the component data has changed
      Designer.Modified;
    end;
  finally
    PEForm.Free;
  end;
end;
```

Ez az eljárás egy olyan formot használ, amit egy különálló egységben definiáltam (azt az egységet ennek be kell építenie a kódjába). A form csak egy Memo komponenst tartalmaz, ahol szerkeszthetjük a számokat, illetve három bitképes nyomógombot, a változtatások elfogadására, visszavonására és a tulajdonság alapbeállításának helyreállítására:

```
object IntListPEForm: TIntListPEForm
  Caption = ' TabString Property Editor'
  object Memo1: TMemo...
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
  object BitBtn3: TBitBtn
    Caption = '&Reset'
    OnClick = BitBtn3Click
  end
end
```

Az egyetlen eljárás, amit meg kell írnunk, az a Reset gomb eseménykezelője, ami az alapértelmezett értékeket másolja a Memo mező Lines tulajdonságába:

```
procedure TIntListPEForm.BitBtn3Click(Sender: TObject);
var
  I: Integer;
```

```
begin
  for I := 0 to 9 do
    Memo1.Lines [I] := IntToStr (I * 100);
end;
```

A lényeg valójában az Edit metódusban található. Ha újra megnézzük az eljárás listáját, láthatjuk, hogy a tulajdonság-szerkesztő a GetComponent metódussal elérheti magát a komponenst, amit szerkeszt (a 0 paraméter azt jelenti, hogy az első komponensre van szükségünk, jelentősége ennek akkor van, ha engedélyezzük, hogy a felhasználó egyszerre több komponens állítson). Amikor elkezdjük szerkeszteni a komponenst, az Edit metódus eléri annak TabsStops tulajdonságát (amely egy tömb). Innen az eljárás elején kiolvassa az eredeti értékeket, majd a végén visszaírja a megváltozottakat.

Megjegyzés

Ha egy komponens egy tulajdonságát tulajdonság-szerkesztővel vagy komponens-szerkesztővel változtatjuk meg, akkor mindig meg kell hívnunk a Designer (tervező) objektum Modified (módosított) metódusát. Ez az objektum általában a formtervező, és a metódus meghívásával azt tudatjuk a Delphi környezettel, hogy adatok változtak meg. A hívás elengedhetetlenül szükséges a fájlmentési műveletek helyes végrehajtásához. Például ha lezárunk egy megváltozott egységet, a Delphi automatikusan rákérdez, hogy menteni akarjuk-e a változtatásokat. Ha elfelejtjük meghívni a Designer objektum Modified metódusát, akkor a komponenseken végzett változtatások elvesznek.

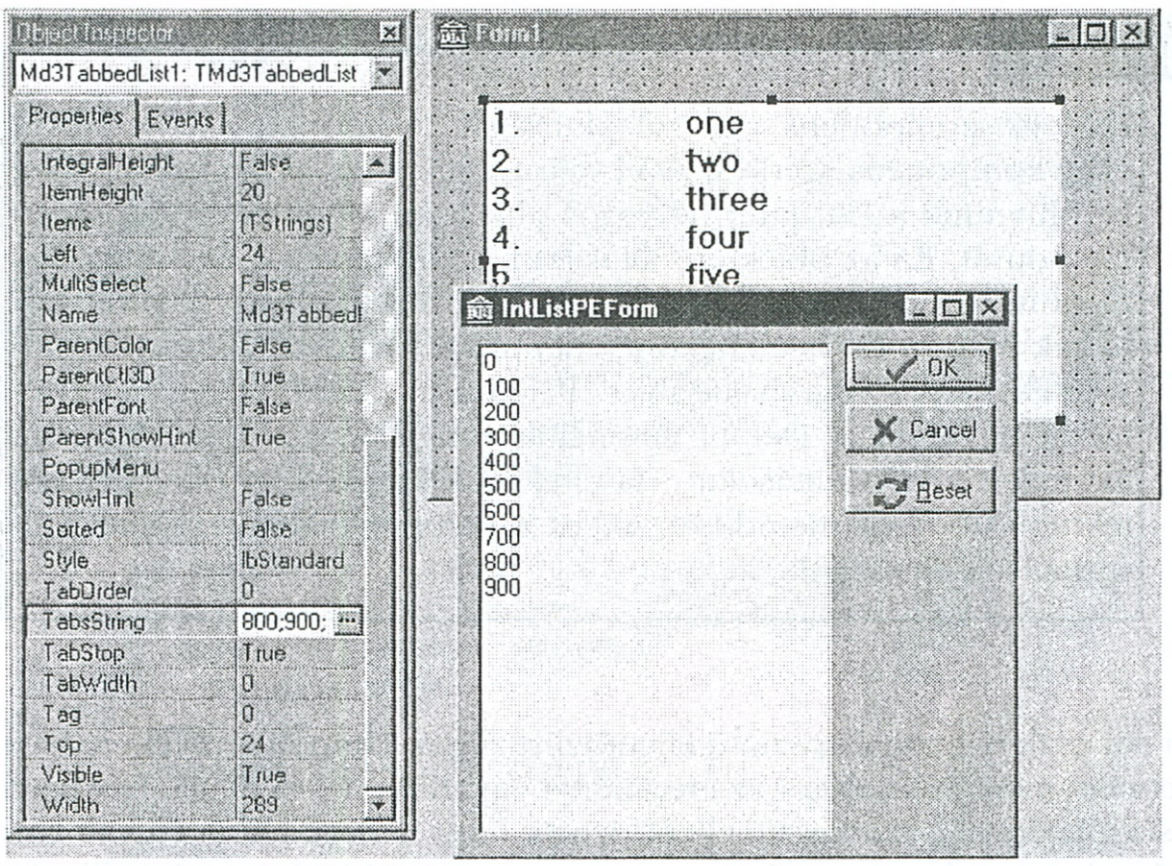
Ez a pár sor csak nagyon alapszintű ellenőrzést végez a kapott értékeken: ha a bemenet nem megfelelő, akkor az eredményt egyszerűen 0-ra állítja. Egy igazi címkelista tulajdonság-szerkesztő ellenőrizheti, hogy a megadott érték valóban csak számokat tartalmazzon (és nem szöveget), ezenkívül megvizsgálhatja, hogy a címkék egymáshoz viszonyítva értelmesek-e (például figyelheti, hogy a címkék növekvő sorrendben következzenek).

A tulajdonság-szerkesztő telepítése

Miután elkészítettük a fenti programot (megtalálható a mellékelt CD-n a fejezet PropEdit mappájában is), a komponens és tulajdonság-szerkesztője telepíthető a Delphibe. Ehhez a következő utasítást kell az egység Register eljárásába írnunk:

```
procedure Register;
begin
  RegisterPropertyEditor (TypeInfo (string),
    TMd3TabbedList, 'TabsString', TMd3IntListProperty);
end;
```

Ezzel a hívással regisztráljuk az utolsó paraméterben megnevezett szerkesztőt string típusú tulajdonságokhoz (első paraméter), de csak egy bizonyos komponenshez és tulajdonsághoz. Ezt a két utóbbi paramétert elhagyhatjuk és ezzel még általánosabb szerkesztőket készíthetünk. Így lehetővé tettük az Object Inspector számára, hogy a komponens tulajdonságait megjelenítse, a felhasználó számára pedig azt, hogy az Edit metódusban definiált párbeszédablak segítségével vegyen fel új címkéket. Ez látható a 19.1. ábrán.



19.1. ábra *A TabStrings tulajdonság az Object Inspectorban és a hozzákapcsolt tulajdonság-szerkesztő*

Ennek a komponensnek a telepítéséhez csak a forrásfájlját kell egy már létező, vagy egy új csomagban elhelyezni. Ahelyett, hogy ezt az egységet és e fejezet további példaprogramjait az Md3Pack csomaghoz adtam volna, inkább egy új csomagot készítettem. Ez tartalmazza a fejezetben szereplő összes bővítőprogramot. Neve: Md3DesPk, ami az Md3 design package (Md3 tervező csomag)

rövidítése. A különbség az eddigi csomagokkal szemben, hogy ezt a {$DESIGNONLY} fordítói direktívával fordítottuk. Ezzel a direktívával jelölhetjük meg azokat a csomagokat, amelyek a Delphi környezettel állnak kapcsolatban, komponenseket vagy szerkesztőket hoznak létre, de az elkészült alkalmazás futtatásakor már nincs rájuk szükség.

Figyeljük meg, hogy a tulajdonság-szerkesztő egysége használja a TabList2 egységet, hiszen a TMd3TabbedList komponens ott lett definiálva. Ezért az új csomagnak hivatkoznia kell a már létezőre. Ezek alapján, az inicializáló kód így néz ki (a fejezet további részeiben újabb egységeket is ide fogunk beépíteni):

```
package Md3DesPk;

{$R *.RES}
{$ALIGN ON}
...
{$DESCRIPTION 'Mastering Delphi 3 DesignTime Package'}
{$DESIGNONLY}

requires
  vcl30,
  Md3Pack;

contains
  PropEdit,
  CompEdit,
  Wizard1,
  IlpeForm;

end.
```

Komponens-szerkesztő készítése

A tulajdonság-szerkesztők használata lehetővé teszi, hogy a programozó még inkább felhasználóbarát komponenseket készítsen. Az Object Inspector a Delphi környezet felhasználói felületének kulcsfontosságú darabja, a fejlesztők folyamatosan használják, ezért a Delphi egy második lehetőséget is biztosít arra, hogy átalakítsuk komponensek rendszerrel való kapcsolattartásának módját: egyéni komponens-szerkesztőket hozhatunk létre.

Ahogy a tulajdonság szerkesztők az Object Inspectort, a komponens-szerkesztők a formtervezőt bővítik ki. Amikor tervezés alatt a formon a jobb egérgombbal kattintunk, az alapértelmezett menüpontokon kívül olyan pontokat is láthatunk, amelyeket a komponens komponens-szerkesztője épített be a menübe. Például ilyen menüpontokkal indíthatjuk el a Menu Designert (menükészítő), a Fields Editort (mezőszerkesztő), a Visual Query Buildert (grafikus lekérdezés-készítő) és a környezet sok egyéb szerkesztőjét is.

Néhány esetben ezeknek a speciális szerkesztőknek a megjelenítése az az alapértelmezett esemény, amit a komponensen való duplakattintáskor végre kell hajtani.

A komponens-szerkesztők általános felhasználási területei közé tartozik a névjegy ablak megjelenítése, amely a komponens készítőjéről adhat információkat, de speciális varázslók elindítását is kezdeményezhetjük innen, melyek a komponens tulajdonságait állítják be.

A TComponentEditor osztály és leszármazottai

A komponens-szerkesztőinket a `TComponentEditor` osztályból kell származtatnunk. Ennek az osztálynak négy virtuális metódusát írhatjuk felül (illetve van még néhány kevésbé fontos, felülírható metódus, de azokat itt most nem nézzük meg):

- A GetVerbCount függvény azt adja meg, hogy hány új menüpontot akarunk a helyi menüben szerepeltetni, ha a formszerkesztőben az adott komponens van kijelölve.
- A GetVerb függvény mindegyik új menüpontra lefut és azt adja vissza, hogy az milyen szöveggel szerepeljen a helyi menüben.
- Az ExecuteVerb eljárás akkor fut le, ha az új menüpontok közül kiválaszt egyet a felhasználó. A menüpont sorszámát kapja meg paraméterként.
- Az Edit metódus hívódik meg, amikor a felhasználó duplán kattint a komponensen a form-tervezőben, és ezzel az alapértelmezett tevékenységet indítja el.

Ha megszokjuk, hogy „Verb" szó semmi mást nem jelent, mint az új menüpontokat és a hozzájuk kötött tevékenységeket, akkor már érthetővé válnak a metódusok elnevezései. Ez az interfész valójában sokkal egyszerűbb, mint amit a tulajdonság-szerkesztőknél már megismertünk.

Komponens-szerkesztő a listához

Most, hogy megismerkedtünk a komponens-szerkesztők készítésének alapötleteivel, itt az ideje, hogy megvizsgáljunk egy konkrét példát. Komponens-szerkesztőnkben meg szeretnék jeleníteni egy névjegy ablakot, egy copyright üzenetet (ez a komponens-szerkesztők egy nagyon gyakori felhasználási területe) és szeretnék elindítani egy speciális tevékenységet, amellyel a címkék beállíthatóak. Ugyancsak el szeretném érni, hogy a komponens alapértelmezett tevékenysége a névjegy ablak megjelenítése legyen egy csipogás után (ennek nem sok értelme van, de jól bemutatja az itt használható technikákat).

Ennek a komponens-szerkesztőnek a megvalósításához a fent felsorolt négy metódust kell felülírnunk:

```
uses
  DsgnIntf;

type
  TMd3TabListEditor = class (TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure ExecuteVerb(Index: Integer); override;
    procedure Edit; override;
  end;
```

Az első egyszerűen visszaadja a helyi menühöz hozzáfűzendő új menüpontok számát:

```
function TMd3TabListEditor.GetVerbCount: Integer;
begin
  Result := 3;
end;
```

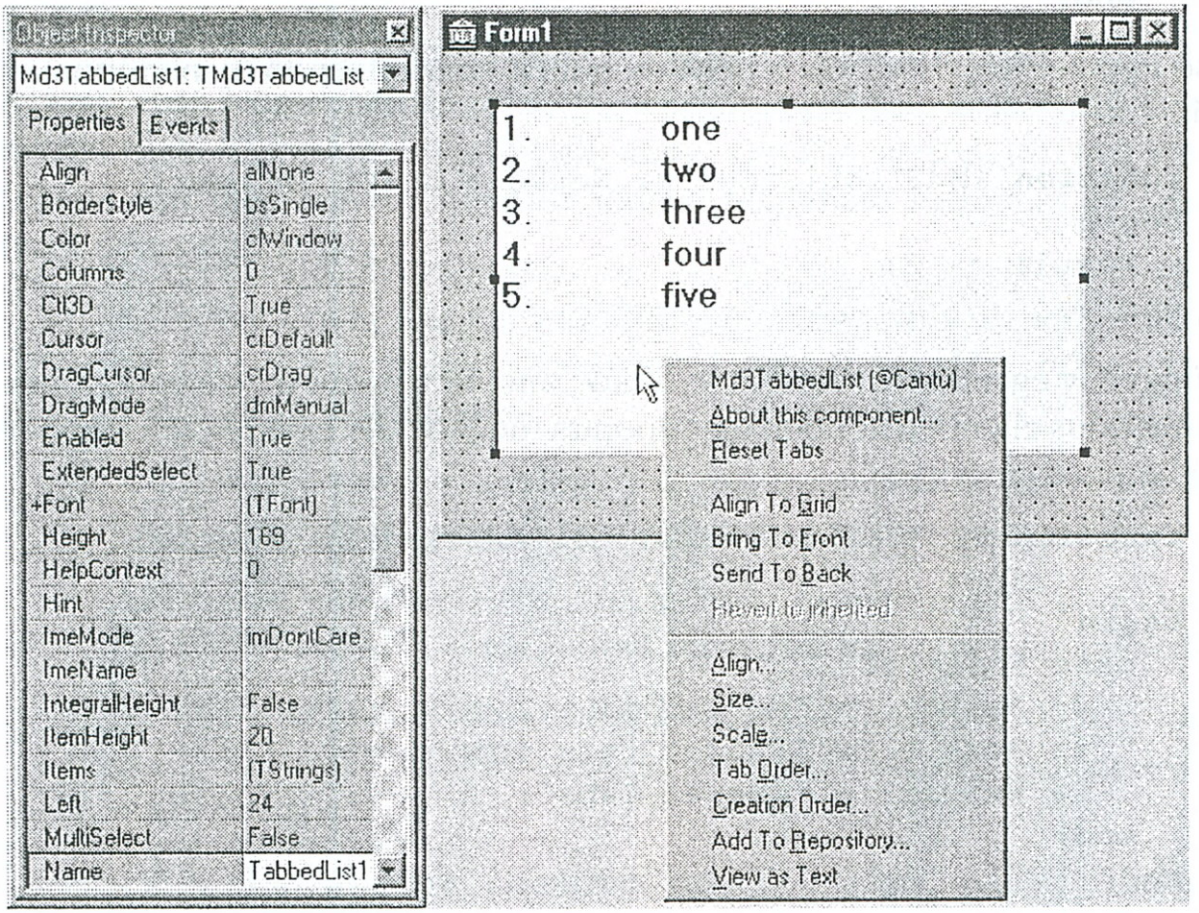
Ez a függvény egyetlenegyszer fut le, akkor, amikor a menü megjelenik. A második metódus ellenben minden menüpontra meghívódik egyszer, tehát ebben az esetben háromszor:

```
function TMd3TabListEditor.GetVerb(Index: Integer):
    string;
begin
  case Index of
    0: Result := 'Md3TabbedList (©Cantù)';
    1: Result := '&About this component...';
    2: Result := '&Reset Tabs';
  end;
end;
```

Ennek a pár sornak az eredményeként a form helyi menüjében megjelennek az új pontok. Ez látható a 19.2. ábrán. Ha ezen új menüpontok közül kiválasztja valamelyiket a felhasználó, akkor a komponens-szerkesztő ExecuteVerb metódusa kerül végrehajtásra:

```
procedure TMd3TabListEditor.ExecuteVerb (
  Index: Integer);
var
  I: Integer;
begin
  case Index of
    0..1: MessageDlg (
      'This is a simple component editor'+ #13 +
      'built by Marco Cantù'+ #13 +
      'for the book "Mastering Delphi 3"',
      mtInformation, [mbOK], 0);
    2: begin
```

```
            with Component as TMd3TabbedList do
              for I := Low (TTabsArray) to High (TTabsArray)
                do TabStops [I] := I * 100;
            // data has changed
            Designer.Modified;
          end;
        end;
      end;
```



19.2. ábra *Az egyéni menüpontok, amelyeket az Md3TabbedList komponens tulajdonság-szerkesztője készített*

Az első két menüpontot egy `case` ágban kezeljük le, és nem írunk külön kódot a copyright üzenetet megjelenítő ponthoz. A megmaradt utasítás megváltoztat néhány adatot a szerkesztett komponens felhasználásával, amit a `TComponentEditor` osztály Component tulajdonságán keresztül érünk el. Ha ismerjük a komponens típusát, akkor tulajdonságait egyszerűen elérhetjük egy típus-átalakítás után (az általános komponensből kell TMd3TabbedListet csinálnunk).

Az utolsó metódus a komponens alapértelmezett tevékenységét állítja be, és akkor fut le, amikor a felhasználó a formtervezőben duplán kattint a komponensen.

```
procedure TMd3TabListEditor.Edit;
begin
  // produce a beep and show the about box
  Beep;
  ExecuteVerb (0);
end;
```

A komponens-szerkesztő regisztrálása

Ahhoz, hogy a Delphi használni tudja szerkesztőnket, regisztrálnunk kell azt. Ehhez megint egy Register eljárást kell az egységébe felvennünk, és abban meg kell hívnunk egy speciálisan komponens-szerkesztőket regisztráló eljárást:

```
procedure Register;
begin
  RegisterComponentEditor (
    TMd3TabbedList, TMd3TabListEditor);
end;
```

Ez az egység is az Md3DesPk csomagba került, ami ennek a fejezetnek az összes programját tartalmazza. Ennek a csomagnak a telepítése és elindítása után készíthetünk egy új projektet, elhelyezhetünk benne egy listát, és kísérletezhetünk programunkkal.

Egyszerű varázslók készítése

A varázslók (wizard, vagy a hagyományos Delphi szakzsargon elnevezésével expert – szakértő) szintén a Delphi bővítési lehetőségei közé tartoznak. A varázslók a Delphi környezet menüihez tudnak új menüpontokat adni, vagy az Object Repository-t bővíthetik. A varázslók célja, hogy a környezet lehetővé tegye egy saját eljárás (a varázsló osztály Execute metódusának) futtatását. A varázsló összes többi eleme – ahogy látni fogjuk – kizárólag arra szolgál, hogy megfelelően telepítsük a Delphi rendszerbe, szabályozzuk képességeit és megjelenését.

Ezek alapján azt gondolhatnánk, hogy a varázslók felhasználási területe nagyon korlátozott. Ez azonban nincs így: a varázslók nagyon hatékonyak lehetnek, mert a Delphi legtöbb funkciójukat objektum-orientált interfészen keresztül teszi elérhetővé, így a környezettel egészen összetett módokon is kapcsolatba léphetünk. Ezek a lehetőségek meghaladják könyvünk kereteit, de azért legalább néhány kulcsfontosságú elemet megvizsgálunk.

A TIExpert osztály és leszármazottai

Az ExptIntf egység definiálja a `TIExpert` osztályt. Ebből kell származtatnunk saját varázslóinkat. Eltérően a fejezetben eddig szerepelt osztályoktól, ez az osztály egy valódi interfész osztály, ezt jelölik kezdőbetűi: TI.

Megjegyzés

A következő két fejezetben látni fogjuk, hogy a Delphiben az interfészekkel az objektumokat megoszthatjuk futtatható fájlok és könyvtárak között. Valójában a Delphi 3 a szokásos komponensekkel is ezt teszi, de a beépített Delphi interfészek (közös nevük ToolsAPI) felépítése ugyanaz maradt, mint a régebbi Delphi változatokban. Figyeljünk rá, hogy az itt szereplő interfész kifejezés hasonló jelentésű, de nem áll közvetlen kapcsolatban az `interface` kulcsszóval, és más COM-hoz kapcsolódó fogalommal, amit a 21. fejezetben kezdünk majd megismerni.

Származtatáskor most is az osztály virtuális metódusait írjuk felül. Habár ez nem feltétlenül szükséges, mégis ajánlott az összes metódust felülírni, amit a `TIExpert` osztályban absztraktnak definiáltak. Ezekről adunk egy nagyon rövid leírást az alábbiakban:

- A GetStyle függvény a varázsló típusát adja vissza. Lehet normál varázsló, ami a Help menüben jelenik meg, lehet form vagy projekt varázsló, amelyek az Object Repository-ban jelennek meg, de lehetnek bővítő varázslók is, melyek bármely Delphi menübe telepíthetik magukat..
- A GetName függvény a varázsló nevét adja meg.
- A GetAuthor függvény a varázsló készítőjét nevezi meg, amit az Object Repository jelenít meg, hogyha azt részletes nézetben használjuk.
- A GetComment függvény a varázsló rövid leírását adja.
- A GetPage függvény az Object Repository azon lapjának a nevét adja vissza, ahová a varázslót telepíteni akarjuk.
- A GetGlyph függvény adja vissza annak az ikonnak az azonosítóját, amit az Object Repository a varázsló megjelenítésére használ.
- A GetState függvény adja meg a varázsló állapotát, ami lehet engedélyezett (kiválasztott) vagy nem az.
- A GetIDString függvény egy olyan karakterlánc azonosítót ad vissza, amely minden varázsló esetén különböző.

- A GetMenuText függvény meghívásával tudja meg a rendszer, hogy mi legyen a menüpont szövege standard varázslók esetén.
- Az Execute eljárás tartalmazza a varázsló valódi kódját.

Egy egyszerű varázsló

A fenti információk alapján elkészíthetjük első, teljesen haszontalan varázslónkat. Semmi mást nem fog tenni, csak megjeleníti az aktuális Delphi projekt nevét, ha van ilyen.

A varázsló osztályok szokásos felépítése a következő:

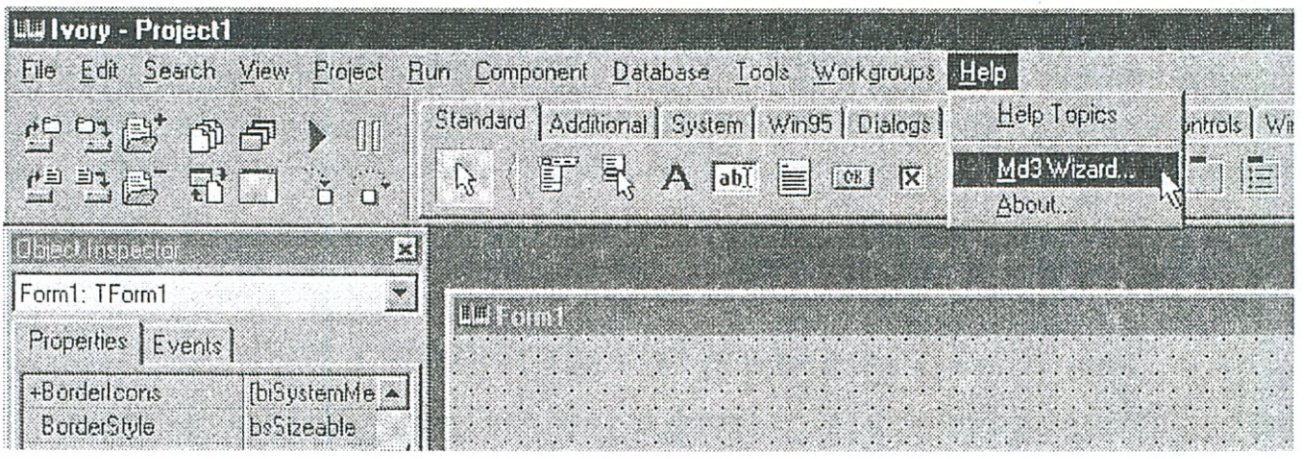
```
uses
  ExptIntf, Windows;

type
  TMd3Expert = class (TIExpert)
  public
    function GetStyle: TExpertStyle; override;
    function GetName: string; override;
    function GetAuthor: string; override;
    function GetComment: string; override;
    function GetPage: string; override;
    function GetGlyph: HICON; override;
    function GetState: TExpertState; override;
    function GetIDString: string; override;
    function GetMenuText: string; override;
    procedure Execute; override;
  end;
```

A fenti metódusok legtöbbjének kódja rettenetesen egyszerű, néhány csak 0-t vagy üres stringet ad vissza. Mások értelmesebb adatot adnának vissza, de soha sem fognak lefutni, mert a lehető legegyszerűbb varázslót készítjük el: a standard varázslót. Ezeket a nem használt metódusokat kivéve lássuk a varázsló programlistáját (az Expert1 mappában ezek kódja is megtalálható a forrásfájlban):

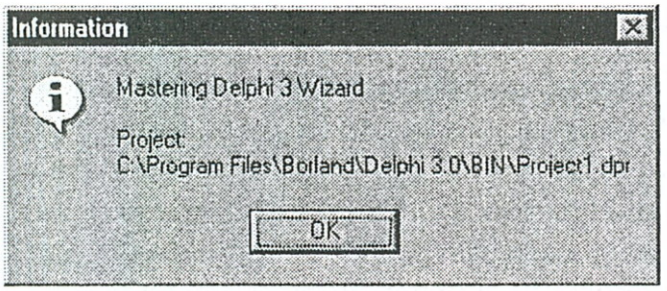
```
function TMd3Expert.GetStyle: TExpertStyle;
begin
  // show up in the Help menu
  Result := esStandard;
end;

function TMd3Expert.GetName: string;
begin
  // official name
  Result := 'Md3 Expert'
end;
```

```
function TMd3Expert.GetState: TExpertState;
begin
  // always enabled, never checked
  Result := [esEnabled];
end;

function TMd3Expert.GetIDString: string;
begin
  // must be unique
  Result := 'MarcoCantu.Md3Expert'
end;

function TMd3Expert.GetMenuText: string;
begin
  // the text of the menu item
  Result := '&Md3 Expert...'
end;

procedure TMd3Expert.Execute;
begin
  // the actual code
  MessageDlg ('Mastering Delphi 3 Wizard'#13#13+
    'Project:'#13 + ToolServices.GetProjectName,
    mtInformation, [mbOK], 0);
end;
```

Ezek a metódusok nem túl sokat produkálnak. Először egy új menüpont jön létre a Delphi Help menüjében (ez látható a 19.3. ábrán). Másodszor, ha kiválasztjuk ezt a menüpontot, akkor a varázsló megjelenít egy üzenetpanelt, benne a Delphi aktuális projektjének nevével (19.4. ábra).



19.3. ábra *A Wizard1 standard varázsló a Delphi Help menüjében hoz létre új menüpontot*



19.4. ábra *A most tárgyalt varázsló által végzett egyszerű tevékenység: mindössze kiírja az aktuális Delphi projekt nevét*

Ennek az egyszerű információnak a megszerzéséhez a varázsló Execute metódusa a beépített ToolServices globális objektum metódusát használja. Ez az objektum a TIToolServices osztály egy példánya, csak tervezés alatt használható és a Delphi környezet egyik elérési pontja. A következő részben rövid leírást adunk erről az osztályról és a ToolsAPI interfész egyéb lehetőségeiről.

A ToolsAPI további lehetőségei

A tulajdonság-szerkesztők, komponens-szerkesztők és a varázslók nem merítik ki a Delphi által nyújtott belső interfészek tárházát, melyekkel további eszközöket fejleszthetünk. Például a Delphiben szerepel egy olyan belső programozói felület, mellyel elkészíthetjük saját vezérlő rendszerünket, saját legördülő menüket adhatunk a Delphi környezethez.

Még sok egyéb beépített felület is szerepel a Delphiben, amelyekkel a bővítménykészítő programozóknak tisztában kell lennie. Mi már használtuk néhány nagyon egyszerű esetben a tulajdonságok Designer objektumát a tulajdonságszerkesztőkben, a beépített ToolServices globális objektumot a varázsló listájában. A következő táblázatban egy rövid összegzést adunk a Delphiben található legfontosabb lehetőségekről. Ezen módszerek részleteinek ismertetéséhez, használatuk bemutatásához példákon keresztül teljes kötetre lenne szükség.

Módszer	Magyarázat
`TFormDesigner`	A tulajdonság-szerkesztők és a komponensszerkesztők számára teszi lehetővé, hogy a form tervezővel, és a jelenleg fejlesztés alatt álló form egyéb komponenseivel kapcsolatban álljanak.
`TIToolServices`	Ezen a felületen keresztül ismerhetjük meg a Delphi állapotát, információkat kaphatunk az aktuális projektről, a szerkesztőkről, a telepített VCL komponensekről, a Delphi környezet menüjének szerkezetéről, a beépített üzenetekről és még sok minden másról is.

Módszer	Magyarázat
`TIModuleInterface`	Ebből az eszközből kiindulva érhetjük el egy egység aktuális információit, szerkesztő felületét, formfelületét (ld később), de itt tudunk egy olyan figyelő eszközt is elindítani, amely a rendszertől érkező üzeneteket lekezelheti, ha a felhasználó valamilyen műveletet végez, például megnyit, elment vagy átnevez egy egységet.
`TIEditorInterface`	A `TIEditReader`, a `TIEditWriter` és a `TIEditView` mellett a Delphi szerkesztővel való kapcsolattartást segíti elő, megnézhetjük, megváltoztathatjuk a szerkesztés alatt álló kódot. Ezen eszközök kezelése különösen bonyolult.
`TIFormInterface`	A `TIComponentInterface` eszközzel együtt a programozó és a tervezés alatt álló Delphi formok közötti kommunikációt bonyolítja le.
`TIStream`	A `TIStream` osztályt és leszármazottait más beépített eljárások fájlok kezelésére használják. Működésük sokban hasonlít a rendes `TStream` osztályokéra.
`TMenuItemInterface`	A Delphi menü menüpontjait érhetjük el ezen az osztályon keresztül, megváltoztathatjuk őket vagy újakat is beilleszthetünk. A `TIMainMenu` interfész segítségével közvetlenül ugorhatunk egy menüpontra.
`TIAddInNotifier`	Ezzel a módszerrel kezelhetünk le olyan globális üzeneteket, amelyek a projekten vagy asztalon történt műveletek hatására keletkeznek.

Tulajdonságok elérése név szerint

Az Object Inspectorban még az általunk elkészített komponensek nyilvános tulajdonságaihoz is hozzáférhetünk. Ezek eléréséhez a rendszer a nyilvános tulajdonságokhoz generált RTTI információkat használja. Néhány új módszer felhasználásával alkalmazásunk elérheti a nyilvános tulajdonságok listáját és használhatja annak elemeit.

Megjegyzés

Alapvetően ezt végzi el az általam készített Object Debugger elnevezésű eszköz. Ez a komponens a mellékelt CD TOOLS mappájában található. Az itt használt ötletek részletes leírását megtalálhatjuk a Delphi Developer's Handbook című könyvben (szerzői: Marco Cantù és Tim Gooch, Sybex, 1997), további részletekkel és példákkal a tulajdonságok név szerinti elérésével kapcsolatban.

Bár kevesen ismerik ezt a lehetőséget a Delphiben, megtehetjük, hogy a tulajdonságokra nevükkel hivatkozunk, és egyszerűen a tulajdonság nevét megadó stringgel kérdezzük le a tulajdonság értékét. A tulajdonságok RTTI információihoz egy pár dokumentálatlan szubrutinnal férhetünk hozzá, melyeket a TypInfo egységben definiáltak.

Megjegyzés

A Borland azért nem dokumentálta ezeket a szubrutinokat, mert a Delphi későbbi verzióiban meg szeretnék változtatni ezek felépítését. Tényleg meg is változtatták őket a Delphi 3-ban a Delphi 1-hez képest, de csak kis mértékben. Azonban ha kihasználjuk ezeket a lehetőségeket, fel kell készülnünk rá, hogy programjainkat csak átalakítással tudjuk majd használni a Delphi későbbi verzióiban.

Ahelyett, hogy végignéznénk a TypInfo egységet, most inkább lássuk azt a rövid programrészletet, amire a tulajdonságok név szerinti eléréséhez szükségünk van. Íme néhány függvény magyarázata, amit a későbbiekben használunk:

A TypInfo egység GetPropInfo függvényével egy mutatót kaphatunk, amin keresztül a tulajdonságról néhány belső információt tudhatunk meg. A mutató típusa `PProfInfo`. A GetPropInfo függvény paramétere ugyancsak egy mutató, amely az osztályról ad belső információkat, és a `TObject` osztály ClassInfo metódusával kérdezhetjük le. Az alábbi programrészlettel egy nyomógomb Caption tulajdonságának információihoz férhetünk hozzá:

```
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo (
    Button1.ClassInfo, 'Caption');
```

A tulajdonság aktuális értékének kiderítéséhez ezután használhatjuk a GetStrProp függvényt, aminek paraméterként az objektumot és a fenti információkra mutató pointert kell megadni:

```
ShowMessage (GetStrProp (Button1, PropInfo));
```

Hasonló függvényekkel érhetjük el más adattípusú tulajdonságok értékeit is, ilyenek például a GetOrdProp vagy a GetFloatProp. Az egész fenti programrészlettel ugyanazt a hatást érhetjük el, mintha a ShowMessage eljárást a `Button1.Caption` paraméterrel hívtuk volna meg. Az egyetlen lényeges különbség, hogy ez a megvalósítás sokkal lassabb, mivel a tulajdonságok elérését a fordító egy jóval hatékonyabb módszerrel végzi. Ezzel szemben a mi

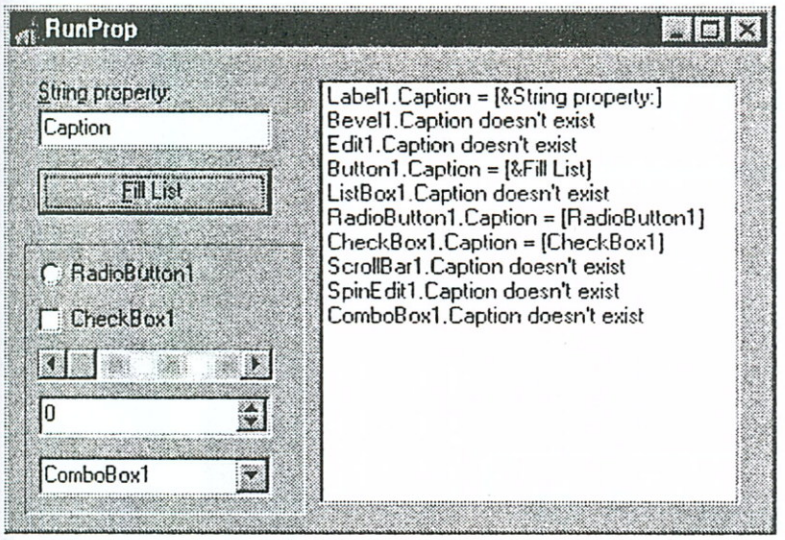
megoldásunk futási időben végzi a hozzáférést. A futás idejű hozzáférés előnye az, hogy sokkal több különböző lehetőségünk lesz programunk megírására, mint ahogy azt a következő, RunProp példában is láthatjuk.

A példa egy listapanelben megjeleníti a formon lévő komponensek egy-egy string típusú tulajdonságát. A vizsgált tulajdonság nevét egy szerkesztőmezőben kérjük be. Ez a programot nagyon könnyen kezelhetővé teszi. A szerkesztőmező és a lista mellett egy nyomógombot is elhelyeztem a formon, amely megnyomására kiszámoljuk az új értékeket. Ezen kívül a formon néhány további komponens is található, csak azért, hogy lekérdezhessük tulajdonságaikat. A gomb lenyomásakor az alábbi eljárás fut le:

```
uses
  TypInfo;
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  PropInfo: PPropInfo;
  Descr: string;
begin
  ListBox1.Clear;
  for I := 0 to ComponentCount -1 do
  begin
    Descr := Components[I].Name + '.' + Edit1.Text;
    // get property RTTI
    PropInfo := GetPropInfo (
      Components[I].ClassInfo, Edit1.Text);
    if PropInfo = nil then
      // if not found, output a message
      ListBox1.Items.Add (Descr + ' doesn't exist')
    else if PropInfo.PropType^.Kind <> tkLString then
      // if not a string, output a message
      ListBox1.Items.Add (Descr + ' is not a string')
    else
      // show the value
      ListBox1.Items.Add (Descr + ' = [' +
        GetStrProp (Components[I], PropInfo) + ']');
  end;
end;
```

A GetPropInfo és a GetStrInfo függvény bemutatásán kívül a metódus két tevékenységet végez el. Ellenőrzi, hogy a GetPropInfo eredménye `nil` érték-e, amivel azt vizsgálja, hogy létezik-e az adott tulajdonság. Ha létezik, akkor ellenőrzi a tulajdonság típusát a `PropInfo.PropType^.Kind` kifejezés segítségével. Ezzel a tulajdonságról néhány belső információt nyerhetünk. A program ezt az értéket a `tkLString` konstanssal hasonlítja össze, amely a hosszú stringeket jelöli. További lehetséges típusok: `tkInteger`, `tkEnumeration`, `tkChar`, `tkSet`, `tkClass`, `tkFloat`, `tkMethod`.

Ha a szerkesztő mezőben az alapértelmezett „Caption" értéket hagyjuk, akkor a Fill List gomb megnyomása után a 19.5. ábrán látható eredményt kapjuk. Más értékeket is kipróbálhatunk, például a „Top", „Text", „Name", „Items" tulajdonságokat, vagy bármely más értelmes beállítást.



19.5. ábra *A RunProp példaprogram kimenete, amely a tulajdonságokat név szerint, futás közben kezeli*

Figyelem!

Ezt a példát nem azért mutattam be, hogy minden más tulajdonság-elérési mód helyett ezt használjuk. Használjuk inkább a szülő osztályok tulajdonság-elérési módszereit, vagy a biztonságos `as` típusátalakítást, ha szükségünk van rá, és csak a lehető legritkább esetben az RTTI-n keresztül történő hozzáférést, hiszen ezzel programunk lelassul, bonyolultabbá válik, könnyen hibázhatunk benne, és nem tudjuk kihasználni, hogy a Delphi fordításkor sok-sok típusellenőrzést elvégez, és programunk hordozhatósága is romlik: valószínűleg a Delphi újabb verzióiban nem fog átalakítás nélkül működni.

Hogyan tovább?

A mellékelt CD-n található néhány általam készített varázsló- és tulajdonságszerkesztő. Néhány ezek közül, mint például a saját komponens varázslóm, sokkal egyszerűbbek, mint az ugyanilyen célra, más cégek által készített kereskedelmi változatok, de azért ezek is hasznosak lehetnek.

Miután megismerkedtünk a komponensekkel és egy kicsit beleástuk magunkat a Delphi mélységeibe, a következő fejezetben a Delphi DLL-jeit vizsgáljuk meg. Már több DLL-lel is találkozhattunk korábbi fejezetekben, tehát itt az

ideje, hogy részletesen megnézzük, mire használhatók, és hogyan készíthetjük el őket. Ez megfelelő alapokat fog szolgáltatni a későbbiekben szereplő komolyabb témákhoz, az OLE és COM programozáshoz. Valójában az eddig bemutatott lehetőségek a Delphiben nagyon hasonlítanak a COM stílusú megközelítéshez, de ezt majd a 21. fejezetben vizsgáljuk meg. Most először nézzük meg a DLL-eket.

20. FEJEZET

Dinamikusan szerkesztett könyvtárak

E fejezet tartalma:

- A DLL-ek szerepe Windows alatt
- A meglévő DLL-ek használata
- DLL létrehozása Delphiben
- Ikonokat tartalmazó DLL készítése
- DLL függvény meghívása futásidőben

Dinamikusan szerkesztett könyvtárak

Windows alatt kétféle futtatható állomány létezik: programok, és dinamikusan szerkesztett könyvtárak (Dynamic Link Libraries, mostantól DLL). Amikor Delphi alkalmazást írunk, akkor általában programfájlt generálunk (EXE-t). Viszont a Delphi alkalmazások gyakran hivatkoznak DLL-ben tárolt függvényekre: minden alkalommal, amikor a programozó egy Windows API-t hív meg, tulajdonképpen egy DLL-re hivatkozik. Míg a Delphi 1 és Delphi 2 verzióknál inkább a statikus szerkesztés volt előtérben a komponensekben és a kódokban, a Delphi 3 lehetővé teszi, hogy a programozó futásidőben használjon DLL-t akár a komponens könyvtárhoz is. Csomagok írásakor tulajdonképpen DLL-t készítünk. A **New** oldal az Object Repository-ban tartalmaz egy DLL váz generátort, amely egy néhány soros forráskódot állít elő.

Delphi környezetben nagyon egyszerű DLL-t készíteni, viszont néhány probléma felmerül ezek természetéből adódóan. Windows alatt DLL-t írni nem mindig olyan egyszerű, mint ahogy első ránézésre annak tűnhet, valamint több futtatható állományt tartalmazó programok (azaz egy főprogram, és egy, vagy több DLL) hibakeresése gyakran szerez kellemetlen perceket a programozónak. Ez a fejezet a DLL programozás alapjait tartalmazza a Delphi szemszögéből. Mint általában, most is mutatok néhány példát arra, hogy miket lehet tenni egy Delphi DLL-be. Mutatok néhány példát más programozási nyelvekből és környezetből is, egyszerűen azért, mert a DLL írásának egyik fő oka éppen az, hogy azt egy más programozási nyelven, más környezetben írt programból is meghívhassák.

A DLL-ek szerepe Windows alatt

Mielőtt belevágnánk a DLL-ek fejlesztésébe Delphiben és más programozási nyelvekben, röviden ismertetem a DLL-ek technikai felépítését, kiemelve a kulcsfontosságú elemeket. Azzal kezdjük, hogy megvizsgáljuk, mi is az a dinamikus szerkesztés, megtekintjük, hogy a Windows hogy használja a DLL-eket, kifejtem, hogy mi a különbség a DLL és a futtaható fájl között, s végül néhány fontos szabályt ismertetek, amiket érdemes követni, ha az ember DLL-t ír.

Mi is az a dinamikus szerkesztés?

Először is meg kell értenie a kedves Olvasónak a függvények és eljárások statikus és dinamikus szerkesztése közötti különbséget. Ha egy szubrutin nem érhető el közvetlenül egy forrásfájlban, akkor a fordító átadja a szubrutint egy belső táblának, ami tartalmazza az összes külső szimbólumot. Természetesen a fordítónak ismernie kell a szubrutin deklarációját, paramétereit és a típusát, ellenkező esetben hibát jelez.

Egy normális statikus programrész lefordítása után a fordító beolvassa a megfelelő Delphi egységből (vagy statikus könyvtárból) a szubrutin lefordított kódját, és hozzákapcsolja a futtatható állományhoz. A keletkező EXE fájl tartalmazza az egész programnak, és a szükséges egységeknek a kódját. A Delphi fordító elég okos ahhoz, hogy a használt egységekből a lehető legkevesebb kódot tegye bele a programba, és csak azokat a függvényeket és eljárásokat szerkessze bele, amelyeket használ a program.

Dinamikus szerkesztés esetében – azaz ha DLL alapú függvényt hívunk meg – a fordító egyszerűen a szubrutin külső deklarációja alapján állítja össze a táblázatokat a futtatható fájlban. Amikor a Windows betölti a futtatható fájlt a memóriába, akkor először a szükséges DLL-eket tölti be, és csak ezután indul el a program. E folyamat alatt a Windows feltölti a program belső táblázatait a függvények memóriabeli címével. Ha valamilyen oknál fogva a Windows nem találja a DLL-t, akkor a program el sem indul.

Minden alkalommal, amikor a program meghív egy külső függvényt, ezt a belső táblázatot használja a hívás továbbításához a DLL-hez (amely most már a program címterületén helyezkedik el). Vegyük észre, hogy ilyenkor nem indul két különböző alkalmazás. A DLL a futó program részévé válik, és minden paraméterátadás a vermen keresztül történik, mint minden más függvényhívásnál.

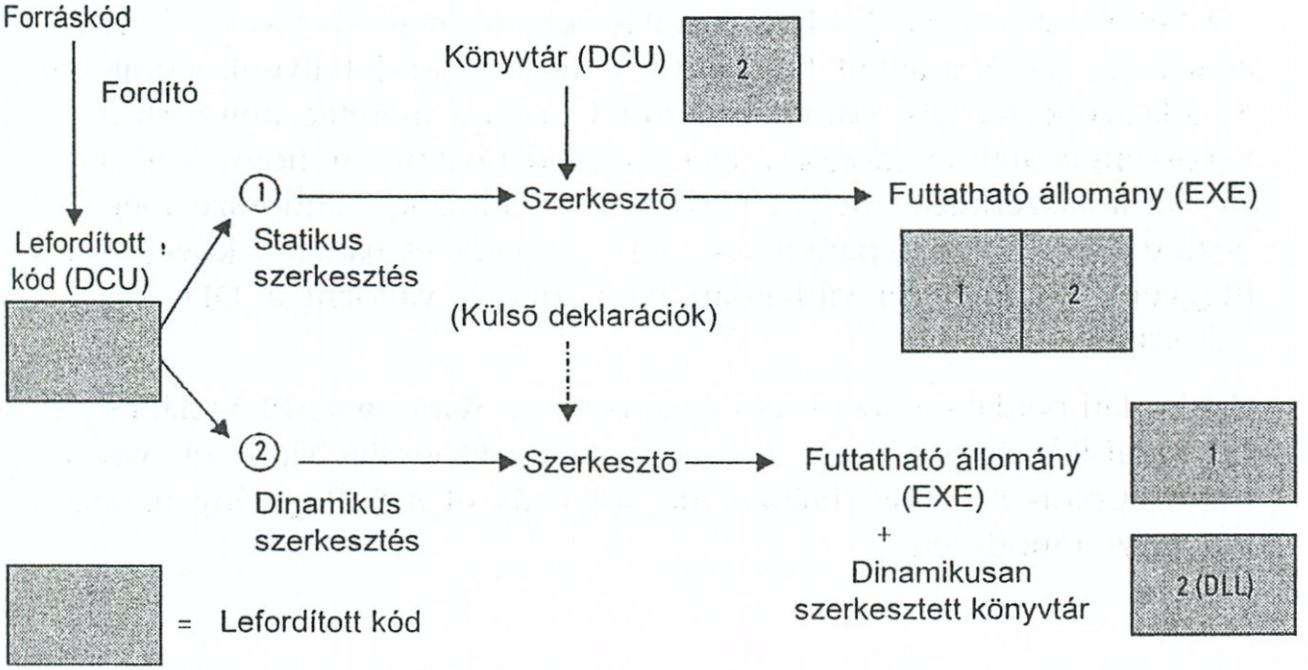
A 20.1. ábra szemlélteti, hogy a program hogyan hívja meg statikusan, és a dinamikusan szerkesztett függvényeket. Felhívom a figyelmet arra, hogy még nem volt szó a DLL fordításáról, mivel először a két különböző szerkesztési mechanizmust akartam elmagyarázni.

Megjegyzés

A dinamikus szerkesztés fogalmának a DLL-ek esetében semmi köze nincs az objektumorientált nyelvek késői kötéséhez. Az Object Pascal-béli virtuális és dinamikus metódusok ugyancsak nem kapcsolódnak a DLL-ekhez. Sajnos azonban ugyanazt a szót használják mindkettőre, ez pedig igen sok bonyodalmat okoz. Amikor ebben a fejezetben dinamikus szerkesztésről beszélek, akkor DLL függvényekre gondolok.

Van egy másik, kevésbé elterjedt megközelítése a DLL-ek használatának, ami még egy fokkal dinamikusabb, mint amelyet fentebb tárgyaltunk. Ebben az esetben az történik, hogy futás közben töltünk be egy DLL-t a memóriába, kikeresünk belőle egy függvényt (feltéve, ha tudjuk a nevét), és meghívjuk. Ez a

módszer komplexebb kódot igényel, és általában lassabb is. Előnye viszont, hogy a DLL-nek nem kell elérhetőnek lennie a program elindulásakor. Ezt a lehetőséget ki is fogjuk használni a DynaCall példaprogramban a fejezet végén.



20.1. ábra *Statikus és dinamikus szerkesztés Windowsban*

Mire való a DLL?

Most, hogy az olvasónak van némi fogalma arról, hogy hogyan működik egy DLL, most már koncentrálhatunk arra, hogy miért érdemes használni. Íme néhány indok, hogy miért használjunk DLL-eket Windows alatt:

- Ha különböző programok ugyanazt a DLL-t használják, akkor a DLL csak egyszer töltődik be a memóriába, ezáltal helyet takarít meg. A DLL-ek az egyes folyamatok (futó alkalmazások) címtartományára képződnek le, de a kódjuk csak egyszer töltődik be a memóriába. A Windows 95 esetében ez a fájl minden folyamat címtartományában ugyanarra a címre kerül.
- Egy adott DLL-t lecserélhető. Ha a DLL-ben levő szubrutinoknak ugyan olyan a paraméterezése, akkor a program lefuttatható az új DLL verzióval, anélkül hogy újra kellene fordítani. Az sem baj, ha a DLL-ben új szubrutinok is vannak. Egyedül az okozhat problémát, ha egy olyan rutin, ami a régebbi verzióban benne volt, hiányzik az újból.

A DLL paraméterek változatai

A legújabb Windows API függvények gyakran egyetlen paramétert kapnak, ami egy pointer arra az adatszerkezetre, ami a tényleges paramétereket tartalmazza. Ezáltal a DLL készítője új paramétereket vehet hozzá az adatszerkezethez anélkül, hogy átírná a meglévő kódot. Ilyenkor tipikusan az adatszerkezet első paramétere tartalmazza a méretet, ami a struktúra verziójára is utal. Ily módon a DLL meg tudja határozni, hogy az alkalmazás az adatszerkezet melyik verziójára hivatkozik, mindössze meg kell néznie a méret/verzió paramétert. Ezt a szemléletet ajánlatos követni, ha a függvény paraméterei várhatóan meg fognak változni a DLL későbbi változataiban.

Gyakorlati példaként az olvasó megnézheti a Windows API általános párbeszédablak függvényeit (például a GetOpenFileName-et vagy a ChooseColor-t) a Súgó fájlban, de sok más új API függvény használja ugyanezt a módszert.

Ezek a kézenfekvő előnyök sok esetben igen hasznosak. Ha van például egy komplex algoritmusunk, vagy egy komplex form, amit több alkalmazás is használ, akkor azokat tárolhatjuk DLL-ben. Ezáltal csökkenthetjük a futtatható programok méretét, és memóriát takaríthatunk meg, ha több programot futtatunk, amelyek ugyanazt a DLL-t használják. A második előny különösen összetettebb alkalmazásoknál használható ki jól. Ha van egy nagyon nagy program, ami rendszeres javítást, és frissítést igényel, akkor feloszthatjuk több futtatható állományra és DLL-re, így elég csupán a megváltoztatott részeket frissíteni. E tulajdonság miatt van értelme a Windows rendszerkönyvtáraknak. Ha a Borland kiad egy újabb verziót a Database Engine könyvtárakból, hogy más SQL szerver adatbázisok is elérhetőek legyenek, akkor nem kell újra lefordítani az alkalmazást, hogy a változásoknak hasznát lássuk.

Egy másik elterjedt technika, hogy a DLL-eket kizárólag erőforrások tárolására használják (a Windows-os erőforrásokat a 27. fejezetben tárgyaljuk részletesen). Létrehozhatunk egy DLL-ből különböző verzióikat úgy, hogy a különböző változatok más-más nyelvű stringeket tartalmazzanak, és így futás közben megváltoztatható a nyelv. Másrészt készíthetünk olyan könyvtárat, ami ikonokat és bitképeket tartalmaz, amiket aztán különböző alkalmazásokból használhatunk.

Mint már említettem is, az egyik kulcsfontosságú előny az, hogy a DLL független a programozási nyelvtől. A legtöbb Windows-os programozási környezetben, beleértve a legtöbb makrónyelvet is, lehetőség van DLL-ben tárolt szubrutin hívására. Ez azt jelenti, hogy készíthetünk DLL-t Delphiben és meghívhatjuk C++-ból, Visual Basic-ből, Paradox-ból, dBase-ből.

A rendszer DLL-jei

A Windows rendszer DLL-ek kihasználják mindazokat az előnyöket, amiket az imént ismertettem. Ezért érdemes közelebbről megvizsgálni őket. Először is, magának a Windows-nak számos rendszer DLL-je van. A Windows három központi része (Kernel, User, GDI) mind DLL-ek felhasználásával készült.

A Windows 95-ben a három fő rendszerkönyvtár megtalálható 16 bites (KRNL386.EXE, USER.EXE, és GDI.EXE), és 32 bites (KERNEL32.DLL, USER32.DLL, és GDI32.DLL) verzióban is. A két verzió gyakran hivatkozik egymásra egy átviteli folyamaton keresztül (angolul ezt thunking néven emlegetik). Windows NT-ben a rendszerkönyvtáraknak csak 32 bites kódjuk van. A többi rendszer DLL csak kiterjesztése az operációs rendszernek, mint például párbeszédablakoknál, DDE-nél, OLE-nél, meghajtóprogramoknál, betűtípusoknál, ActiveX kontroloknál, és sok még sok más esetben.

Magában a Windows-ban a DLL-ek használata rendkívül fontos. Tulajdonképpen az operációs rendszer egyik leglényegesebb technikai alapjai a DLL-ek. Mivel minden alkalmazás használja a rendszer DLL-eket (például ablak létrehozására), ezért minden program kapcsolatban áll ezekkel a DLL-ekkel. Vessünk egy pillantást arra, hogy miért is lehet egy könyvtárnak több verziója.

Nézzük először a meghajtóprogramokat: ha kicseréljük a nyomtatót, akkor nem kell újraírni az alkalmazást, vagy új verziót venni a Windows GDI könyvtárból, ami a nyomtató kimenetet vezérli. Ahhoz, hogy elérhessük a nyomatót, mindössze egy adott meghajtót kell telepíteni, ami nem más, mint egy olyan DLL, amit a GDI hív meg. Minden nyomtatótípusnak megvan a maga meghajtója, azaz DLL-je, ezért a rendszer rendkívül rugalmas.

Másrészt pedig a verziók kezelése nagyon fontos a rendszer számára. Ha adott egy Windows 3.1 alatt lefordított alkalmazás, akkor az futtatható kell hogy legyen Windows 95 alatt is. A Windows minden verziójának különböző rendszerkódja van (és a Windows 95 16 bites támogatása tulajdonképpen kijavítja néhány hibáját a Windows 3.1-nek), de mivel minden újabb verzió tartalmazza a régebbi API függvényeket, a régi kód továbbra is működik, bár nem tudja kihasználni az újabb API függvényeket. Viszont a régi kód is kihasználhatja az újabb verzió előnyeit, ha valamelyik létező függvény megváltozik. Az egyik kézenfekvő példa a felhasználói felület: ha készítünk egy alkalmazást Windows 3.1-re, akkor futtathatjuk Windows 95 alatt, és automatikusan megváltozik a felhasználói felület. Nem kellett újrafordítani az alkalmazást, mert a program magától kihasználja a dinamikusan hozzákapcsolt rendszerkönyvtár új tulajdonságait.

Szokás a rendszer DLL-eket bizonyos rendszerinformációk archívumaként használni. Például a USER.DLL tartalmaz egy listát a rendszerben lévő összes aktív ablakról, a GDI.DLL pedig az aktív tollak, ecsetek, ikonok, bittérképek

listáját tartalmazza. E két rendszer DLL szabad memóriaterületét általában a „szabad erőforrások" néven emlegetik, és – amint azt részletesen láthatjuk majd a 26. fejezetben – ezeknek nagy jelentőségük van a Windows-ban.

Csomagok (package-ek) statikus és dinamikus használata

Tipikus Delphi 3-as DLL-ek a komponenscsomagok. A csomagok lehetővé teszik, hogy több komponenset összerakjunk, majd összekapcsoljuk a komponenseket akár statikusan (azaz hozzáadva a kódjukat az alkalmazásunk futtatható állományához), akár dinamikusan (azaz DLL-ben tartva a komponens kódját). Az előző fejezetben már láttuk, hogy hogyan kell csomagot építeni. Most szeretném a futásidejű csomag szerkesztés két fajtájának előnyeit és hátrányait ismertetni. Sok mindent szem előtt kell tartani:

- Ha egy csomagot DLL-ként használunk, akkor a futtatható állományok sokkal kisebbek lesznek. A statikusan szerkesztett csomag lehetővé teszi, hogy a kódjának csak bizonyos részeit frissítsük. Általában egy alkalmazás futtatható állományának mérete, és a hozzá tartozó DLL-ek mérete együtt sokkal nagyobb, mint a statikusan szerkesztett programoké.
- Ha több olyan Delphi alkalmazást juttatunk el a felhasználókhoz amelyek ugyanazon a csomagon alapulnak, akkor lehet, hogy a végén kevesebb kódot kell szállítani.
- Több Delphi alkalmazást futtatva, amelyek ugyanazon a csomagon alapulnak, megtakaríthatunk némi memóriaterületet.
- Ha egy csomagot DLL-ként használunk, akkor ha van egy új, frissített változata a csomagnak, akkor nem kell újrafordítani a programot, hanem egyszerűen csak telepíteni a DLL új verzióját.
- Ne felejtsük el, hogy amikor apróbb változtatásokat végzünk a programban, akkor használhatunk frissítő programot is, így elegendő csak a változásokat tartalmazó fájlt terjeszteni, így nem kell az összes fájlból új példányt telepíteni.

Különbségek a DLL-ek és az EXE-k között

Most, hogy ismerjük az alapvető elemeit a dinamikus szerkesztésnek és tudunk néhány indokot, amiért használni érdemes, most már koncentrálhatunk arra, hogy mi is a különbség egy normális futtatható állomány (EXE fájl) és egy dinamikusan szerkesztett könyvtár (DLL fájl) között. Az EXE fájlok és a DLL-ek belső szerkezete többnyire azonos. Akkor érzékelhető az első különbség, amikor a DLL betöltődik a memóriába. Amint már említettem, a Windows a DLL

kódját csak egyszer tölti be a memóriába. Ugyanez történik a futtatható állományokkal is, akkor is, ha több példányban indítjuk el őket. Mindkét esetben egy számláló mechanizmus gondoskodik arról, hogy a kódot kidobja a rendszer a memóriából, amikor az összes őt használó program terminál.

A legfőbb különbség a programok és a DLL-ek között az, hogy a DLL-ből nem lesz futó program, amikor betöltődik a memóriába. A DLL mindössze eljárások és függvények gyűjteménye, amelyeket a futó programok meghívhatnak. Ezek az eljárások és függvények a hívó program vermét használják. Így tehát a másik fő különbség a program és a könyvtár között az, hogy a könyvtár nem készít vermet, hanem a hívó programét használja. A Win32 esetében a DLL minden memóriaigényét a hívó folyamat címtartománya elégíti ki, mivel a DLL az alkalmazás címtartományába töltődik be.

A DLL készítésének alapszabályai

Az eddigiek összefoglalhatók néhány szabály formájában, amelyeket érdemes szem előtt tartania minden DLL programozónak. Egy olyan DLL függvénynek, illetve eljárásnak, amit külső programok hívnak meg, meg kell felelnie az alábbi kritériumoknak:

- Szerepelnie kell a DLL export részében levő listában. Ez teszi láthatóvá a rutint a külvilág számára.
- Delphi 3-ban az exportált függvényeket `stdcall`-ként is deklarálni kell, ezzel állítjuk be, hogy a szabványos Win32 paraméterátadási technológiát használja a rendszer, az optimalizált regiszteres paraméterátadási technológia helyett (ez utóbbi az alapértelmezett a Delphiben).
- A Win32-ben egy DLL használhat úgy globális adatot, hogy nem osztja meg a hívó alkalmazással. Amikor egy alkalmazás betölt egy DLL-t, a DLL globális adatait a saját címtartományába tölti be (ezt látni fogjuk a DllMem példában a későbbiekben).

A Win16 és a Win32 DLL-ek

A másik lényeges tulajdonsága a DLL-eknek az, hogy két fajtájuk van. Vannak Windows 3.1 (Win16) illetve Windows NT vagy Windows95 (Win32) DLL-ek. A Delphi 16-bites verziójával írt könyvtárak az első csoportba tartoznak, a 32-bites változattal fordított könyvtárak a másodikba.

Mint már említettem, a 16 és a 32-bites DLL-ek sajnos nem kompatíbilisek. Például nem lehet 16 bites DLL-t hívni Delphi 2 programból. Ez nem a Delphi gyengesége, hanem egy tipikus Windows-os probléma. Tulajdonképpen van is rá megoldás: használhatjuk a Microsoft megfelelő alkalmazását a konverzió elvégzésére. Ezt teszi a Windows 95 is, amikor 16-bites rendszerkönyvtárakat hív meg 32 bites alkalmazásokból, vagy amikor 32-bites rendszerkönyvtárakat

hív meg régi 16-bites alkalmazásokból. Az átvivő mechanizmus használata igen bonyolult és lassú, használatához szükség van egy specifikus alkalmazásra (egy úgynevezett thunk compiler-re), ezért aztán ezt a témát a továbbiakban nem is tárgyaljuk.

A meglévő DLL-ek használata

Mielőtt nekiállnánk Delphiben DLL-eket írni, érdemes tisztázni, hogy hogyan is kell használni őket. Már láttunk erre több példát is az előző fejezetekben, amikor Windows API függvényeket hívtunk meg. Talán emlékszik még a kedves Olvasó, hogy minden API függvény a Windows rendszeregységében van. Az egység `interface` részében a következőképpen definiálták a függvényeket:

```
function PlayMetaFile(DC: HDC; MF: HMETAFILE): BOOL;
    stdcall;
function PaintRgn(DC: HDC; RGN: HRGN): BOOL; stdcall;
function PolyPolygon(DC: HDC; var Points; var nPoints;
  p4: Integer): BOOL; stdcall;
function PtInRegion(RGN: HRGN; p2, p3: Integer): BOOL;
    stdcall;
```

Az implementációs részben pedig a kódjuk helyett az egység egy DLL-ben levő külső definícióra hivatkoznak:

```
const
  gdi32 = 'gdi32.dll';
function PlayMetaFile; external gdi32 name
    'PlayMetaFile';
function PaintRgn; external gdi32 name 'PaintRgn';
function PolyPolygon; external gdi32 name
    'PolyPolygon';
function PtInRegion; external gdi32 name 'PtInRegion';
```

Ezeknek a külső függvényeknek a definíciója az általuk használt DLL nevére hivatkozik. A DLL nevének tartalmaznia kell a .DLL kiterjesztést, különben a program csak Windows 95 alatt fog működni, NT alatt nem. A másik elem magának a DLL függvénynek a neve. A `name` direktíva elhagyható, ha a Pascal függvény (vagy eljárás) neve megegyezik a DLL függvény nevével.

Ha egy DLL-ben levő függvényt akarunk meghívni, akkor külön is megadhatjuk a deklarációját, és a külső definícióját (mint a fenti példában láttuk), de a kettőt egyszerre is elintézhetjük egyetlen deklarációban. Ha egyszer egy függvényt megfelelően definiáltunk, akkor a Delphi alkalmazásunk kódjában ugyanúgy meghívhatjuk, mint bármely más függvényt. Nincs semmi különleges a hívás szintaxisában sem. Ugyanolyan függvény illetve eljáráshívás, mint bármely másik.

Írtam egy nagyon egyszerű DLL-t C++-ban néhány haszontalan függvénnyel, hogy azon keresztül mutathassam be, hogy miként kell függvényeket hívni Delphiből. Nem fogom kommentálni a C++ kódot (különben is alapjában véve C kód), viszont részletesebben tárgyalom majd a Delphi alkalmazások és a C++ DLL-ek közötti hívásokat. Delphi programozás során egyébként is gyakran használunk C-ben, vagy C++-ban írt DLL-eket.

Megjegyzés

Amióta megjelent a Borland C++ Builder (a C++ nyelvre épülő Delphi „klón") azóta a C++ és az Object Pascal alkalmazások közötti kódmegosztás lehetősége exponenciálisan növekszik. A C++ Builder közvetlenül olvas be Pascal egységeket és használ Delphi komponenseket. Itt most egy egyszerűbb, és hagyományosabb megközelítést mutatok be.

DLL készítése C++-ban

Az alábbi programlista a CppDll könyvtár felépítéséhez használt C++ fájl forráskódja. Vélhetően ezt a C++ kódot bármilyen C++ fordítóval le lehet fordítani, bár csak Borland fordítókkal teszteltem (Borland C++ 5 és C++ Builder). Íme a könyvtár teljes C++ kódja:

```
#include <windows.h>

extern "C" __declspec(dllexport)
int WINAPI Double (int n)
{
    return n * 2;
}

extern "C" __declspec(dllexport)
int WINAPI Triple (int n)
{
    return n * 3;
}

__declspec(dllexport)
int WINAPI Add (int a, int b)
{
    return (a + b);
}
```

Ez a kód igen egyszerű. A három függvény néhány elemi számolást végez el a paraméterekkel, majd visszaadja az eredményt. Vegyük észre, hogy minden függvényt a WINAPI módosítóval definiáltunk, amely beállítja a megfelelő paraméterátadási konvenciót, valamint minden függvényt a `__declspec (dllexport)` deklaráció előz meg. Ez utóbbinak az a feladata, hogy láthatóvá tegye a függvényt a külvilág számára.

Ezek közül a C++ függvények közül kettő a C-ből ismert elnevezési konvenciót követi (ezt jelzi az `extern "C"` kifejezés), de a harmadik nem. A továbbiakban megvizsgáljuk, hogy ez miként befolyásolja a függvények hívási módját. A fenti fájl az egyetlen forrásfájl a CppDll projektben. Egyszerűen töltsük be egy C++ környezetbe (vagy nyissuk meg projektként a C++ Builderbe), állítsuk be a megfelelő projekt opciókat, hogy DLL-re legyen a fordítás eredménye. Természetesen használhatjuk a CD mellékleten lévő DLL fájlt és koncentrálhatunk az eredeti témánkra, azaz olyan Delphi program írására, ami C++ DLL-t hív meg.

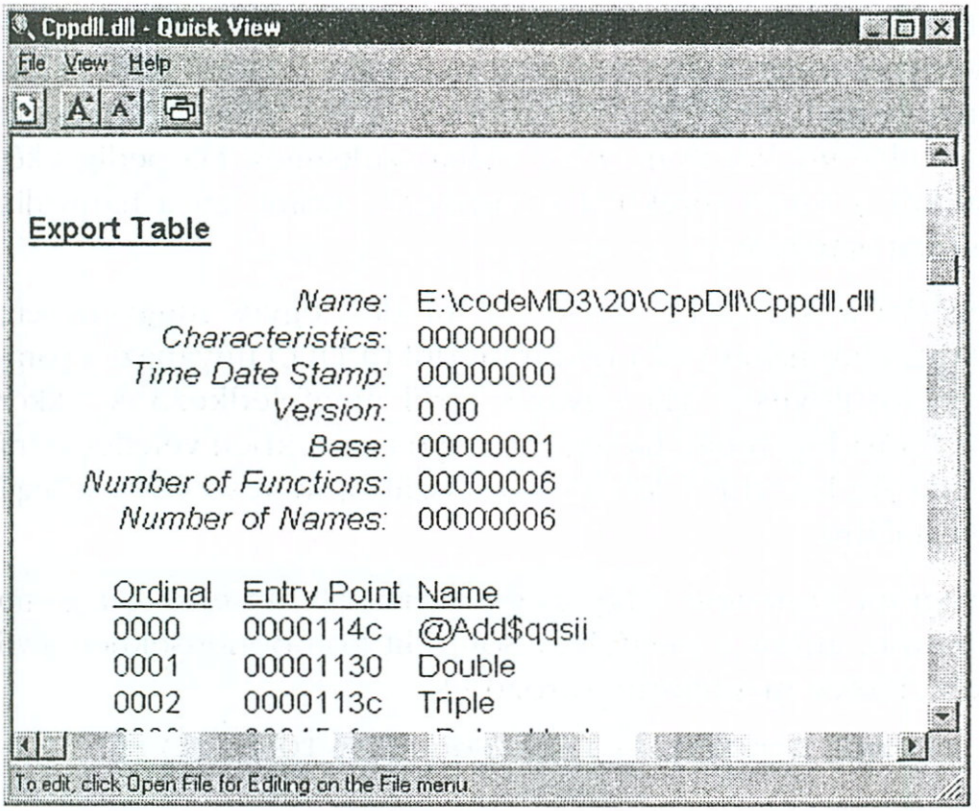
C++ DLL függvény deklarálása Delphiben

Most, hogy megírtuk ezt a DLL-t C++-ban, elkészíthetünk egy Delphi programot, ami meghívja azt. A CD mellékleten a CPPDLL könyvtárban van egy nagyon egyszerű CallCpp nevű példa. A formja három gombot tartalmaz, amelyekkel a DLL egyes függvényeit lehet meghívni, valamint két `SpinEdit` komponenst a paramétereknek, és egy csak olvasható keretet az eredmény kijelzésére.

Hogy ezt a programot megírhassuk, először is a függvényeket megfelelő módon definiálni kell. Ehhez először nézzünk bele a DLL-be, hogy lássuk az exportált függvények neveit. Ha Windows 95-öt használunk és telepítettük a futtatható fájlokhoz tartozó böngészőt (ami opcionális része a rendszernek), akkor egyszerűen csak az egeret a DLL fájlra állítjuk, megnyomjuk a jobb gombot, majd kiválasztjuk a Quick View (nézőke) menüpontot. A megjelenő ablak néhány mélyebb szintű technikai információt tartalmaz a futtatható fájlról. Minket most az `Export Table` rész érdekel. Ezt a 20.2. ábra illusztrálja.

Megjegyzés

Vegyük észre, hogy mind a három függvénynek neve és sorszáma is van (felsorolásként jelölve). Ezt az indexelést eredetileg Windows 3.1 függvényekhez találtak ki. A Win32 szabvány szerint a Microsoft azt javasolja, hogy a függvényekre név szerint hivatkozzunk.



20.2. ábra *A Windows 95 Nézőkéje (Quick View) lehetővé teszi a DLL és az EXE fájlok vizsgálatát. A képen a CPPDLL.DLL fájl export táblája látható*

A Nézőke helyett használhatjuk a TDUMP32 parancssorból futtatható programot, ami a Delphi része. Ez még részletesebb információt ad a futtatható fájl belső szerkezetéről.

Az Add függvény kivételével a függvények belső nevei megegyeznek a C++ forrásfájlban levő nevükkel. Mivel nem használtuk az `extern "C"` kifejezést az Add függvénynél, a C++ fordító automatikusan hozott létre nevet a függvénynek, úgy, hogy a paraméterek számára és típusára vonatkozó információból készíti azt el. Ezt a technológiát igényli a C++ nyelv a függvények betöltéséhez. Az eredmény egy furcsa függvénynév: `@Add$qqsii` (Borland C++ fordító esetében). Erre a névre kell hivatkoznunk a Delphi példánkban, ha meg akarjuk hívni az Add DLL függvényt (ez magyarázat arra, hogy miért kerülendő a C++ automatikus névgeneráció az exportált függvényeknél, és miért ajánlatos mindegyiküket `extern "C"`-vel deklarálni).

Itt következik három függvény deklarációja a Delphi CALLCPP példából:

```
function Add (A, B: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name '@Add$qqsii';
function Double (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL ' name 'Double';
function Triple (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL ';
```

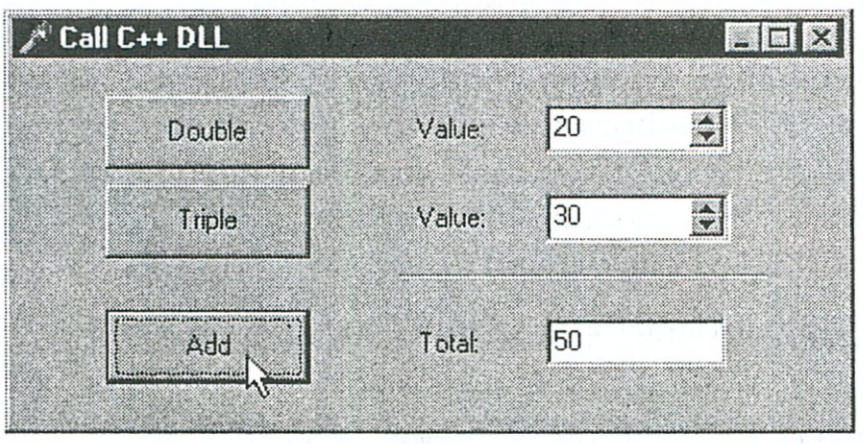
Amint látható, a külső függvénynek adhatunk álnevet (alias), de ez nem kötelező. Az első függvénynél megtettem (nem volt más választásom, mert az exportált DLL függvénynév nem megengedett azonosító a Pascalban). A második függvénynél is adtam álnevet, bár itt nem lett volna szükséges. Ha pedig a két név megegyezik, akkor a `name` direktíva elhagyható, amint azt a harmadik függvény esetében meg is tettem.

Ne felejtsük el megadni a `stdcall` direktívát minden egyes függvényben, hogy a hívó modul (az alkalmazás) és a hívott modul (a DLL) ugyanazt a paraméterátadási konvenciót kövesse. Ha ugyanis erről megfeledkezünk, akkor nem mindig kapunk futásidejű hibát, hanem bizonyos esetekben véletlenszerű függvényhívás következik be, ami a lehető legrosszabb, mert az ilyen jellegű hibát igen nehéz megtalálni.

A CallCpp példaprogram kimenete nagyon egyszerű. Ha az első két gomb valamelyikére kattintunk, akkor a megfelelő SpinEdit komponensekben lévő érték megduplázódik, illetve megháromszorozódik:

```
procedure TForm1.BtnDoubleClick(Sender: TObject);
begin
  SpinEdit1.Value := Double (SpinEdit1.Value);
end;

procedure TForm1.BtnTripleClick(Sender: TObject);
begin
  SpinEdit2.Value := Triple (SpinEdit2.Value);
end;
```

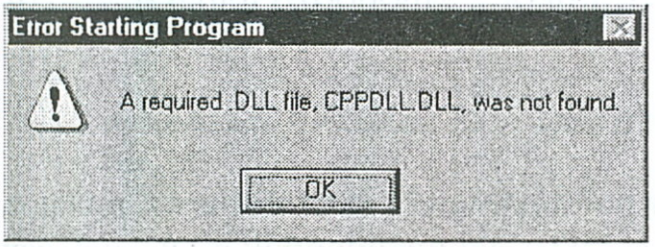
Ha a harmadik gombra kattintunk, akkor a két SpinEdit komponensben lévő érték összeadódik, a harmadik DLL függvény meghívásával pedig az összeg megjelenik az szerkesztőmezőben. Láthatunk példát is a program működésére a 20.3. ábrán. Íme a harmadik gomb OnClick eseménykezelőjének a kódja:



20.3. ábra *A CallCpp példaprogram mindhárom gomb megnyomása után*

```
procedure TForm1.BtnAddClick(Sender: TObject);
begin
  Edit1.Text := IntToStr (Add (
    SpinEdit1.Value, SpinEdit2.Value));
end;
```

Vegyük észre, hogy ahhoz, hogy ezt az alkalmazást futtathassuk, a DLL-nek ugyanabban a könyvtárban kell lennie, mint a projektnek, esetleg a PATH környezeti változó által mutatott helyen, vagy a Windows, vagy a System könyvtárak valamelyikében. Ha a futtatható állományt új könyvtárba helyezzük át, és megpróbáljuk futtatni, akkor vélhetően futásidejű hibaüzenetet kapunk, ami azt jelzi, hogy a DLL hiányzik. Ezt illusztrálja a 20.4. ábra.



20.4. ábra *A hibaüzenet, amelyet a Windows akkor ad, ha nem találja a hivatkozott DLL-t*

DLL létrehozása Delphiben

A Delphiben alkalmazhatunk más környezetekben írt DLL-eket is, de használható arra is, hogy DLL-eket építsünk, amikre aztán hivatkozhatunk Delphi programokból, vagy más olyan környezetből, amelyek támogatják a DLL-t. A Delphiben olyan egyszerű a DLL-ek készítése, hogy az ember hajlamos egy kicsit túlzásba vinni a használatukat. Általában azt tanácsolom, hogy inkább komponenseket és csomagokat építsünk, mint sima DLL-eket. A csomagok olyan komponenseken alapulnak, amelyek lehetővé teszik, hogy objektum-orientált kódot írjunk, majd hatékonyan újra és újra felhasználjuk azt. Ezzel szemben a hagyományos programozási szemléletet követve azt is megtehetjük, hogy egyszerűen egy eljárásgyűjteményt helyezünk el a DLL-ben.

Amikor DLL-t írunk, akkor általában szubrutinokat, függvényeket, és eljárásokat exportálunk, bár látni fogjuk a következő fejezetben, hogy a Delphi lehetővé teszi osztálymetódusok exportálását is. Az osztálymetódusok exportálása a COM-mal van szorosabb kapcsolatban, amelyet a következő fejezetben tárgyalunk.

Általában, amikor összetettebb Delphi alkalmazásokat építünk, akkor objektum-orientált programozási technikákat használunk az alkalmazás

szerkezetének definiálására. Ha a későbbiekben szétosztjuk az alkalmazások kódját a DLL-ek között, akkor ezt az előnyt elveszítjük.

Ha ugyanazokat a függvényeket különböző környezetekből is meg kell hívni, akkor érdemes kis függvényekből álló könyvtárakat létrehozni. Azaz írhatunk DLL-t fordítóval rendelkező nyelven is, például Object Pascalban, és meghívhatjuk őket interpreteres környezetből is. Persze, ha lehetőség van rá, akkor ajánlatos az egész programot Delphiben megírni.

Mint már említettem, akkor is hasznos lehet a DLL-ek készítése, ha a program kódjának egy részét rendszeresen át kell írni. Ilyenkor úgy cserélhetjük ki a DLL-t, hogy a program többi része változatlan marad. Hasonlóképpen járhatunk el, ha olyan programot kell írni, amely különböző felhasználói csoportoknak különböző szolgáltatásokat nyújt. Ilyenkor a különböző felhasználókhoz különböző DLL-eket kell eljuttatni.

Az első egyszerű Delphi DLL

Mielőtt egy igazi példát kezdenénk el tanulmányozni (DLL formot), mutatok egy nagyon egyszerű, Delphiben épített DLL-t. Ez ugyanannak a DLL-nek egy újabb verziója, amit az imént írtunk C++-ban. E példán elsősorban a Delphi alatt fejlesztett DLL-ek definiálásának szabályait kívánom bemutatni. Először is válasszuk ki a **File** menüből a **New** parancsot, majd a DLL opciót az Object Repository **New** lapján. Ez egy nagyon egyszerű forrásfájlt készít, amely az alábbi definícióval kezdődik.

```
library Project1;
```

A `library` kifejezés azt jelzi, hogy nem futtatható állományt, hanem DLL-t akarunk építeni. Most már tehetünk rutinokat a könyvtárba, és felsorolhatjuk őket az `exports` kifejezésben:

```
function Triple (N: Integer): Integer; stdcall;
begin
  Result := N * 3;
end;

function Double (N: Integer): Integer; stdcall;
begin
  Result := N * 2;
end;

exports
  Triple, Double;
```

Ilyen egyszerű DLL esetében nincs szükség a `uses` utasításra. Egy kicsit megváltoztattam a kódot az előző verzióhoz képest, hogy minden függvényhívásnál megjelenjen egy üzenet. Kétféleképpen lehet üzenetet megjeleníteni. A legegyszerűbb az, ha a következőképpen módosítjuk a kódot:

```
uses
  Dialogs;

function Triple (N: Integer): Integer; stdcall;
begin
  ShowMessage ('Triple function called');
  Result := N * 3;
end;
```

Ezzel a módszerrel a Delphi beépít rengeteg VCL kódot az alkalmazásba. Ha statikusan szerkesztjük a VCL-t, akkor a méret 197 kilobájt lesz. Jobb megoldás, ha az üzenetet direkt API híváson keresztül jelenítjük meg, mert akkor nincs szükség a VCL kódra.

```
uses
  Windows;

function Triple (N: Integer): Integer; stdcall;
begin
  MessageBox (0, 'Triple function called',
    'First DLL', mb_OK);
  Result := N * 3;
end;
```

Ha a kódot így módosítjuk, akkor az alkalmazás mérete lecsökken 14 kilobájtra. Ez a hatalmas különbség nyilván intő példa a DLL-ek túlzott használata ellen. A példa forráskódjában, a CD mellékleten a könyvtár mindkét verziója megtalálható, s az egyiket megjegyzésekkel is elláttam. Ezt a részt megváltoztatva könnyen átírható a kód, így kedvére kísérletezhet az olvasó.

Delphi DLL hívása

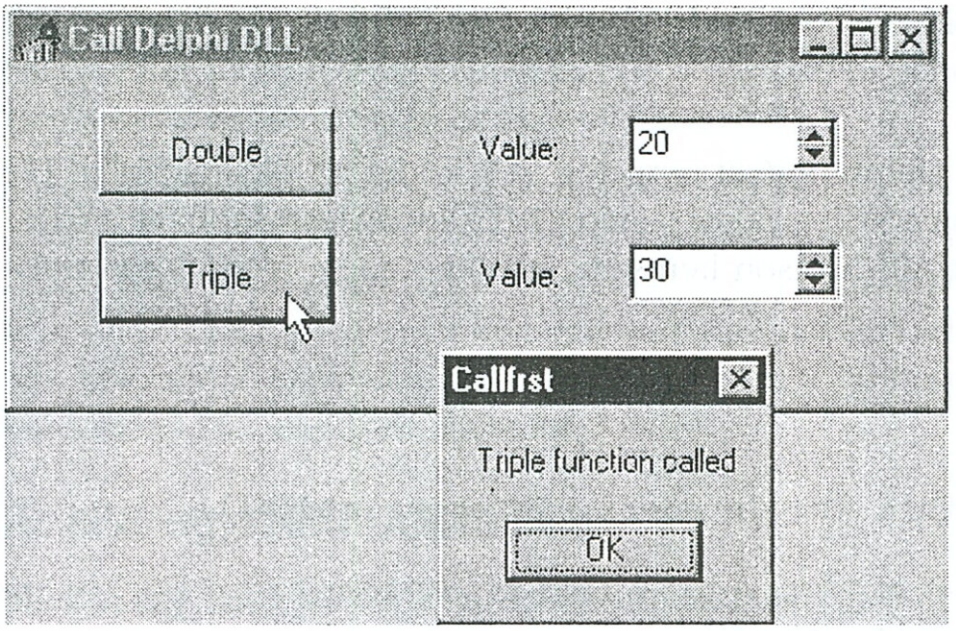
Hogyan lehet használni ezt a könyvtárat? Meghívható egy másik Delphi projektből, de más környezetből is. Példaként elkészítettem a CallCpp példa egy újabb változatát, elneveztem CallFrst-nek és a CD melléklet FirstDLL könyvtárába raktam. Ez az újabb példa az előző példa formjának a felső felét tartalmazza, az Add gomb és a csak olvasható szerkesztőmező nélkül.

Hogy elérhessem a DLL függvényeket, deklarálnom kell őket külső függvényként, ugyanúgy ahogy a C++ DLL esetében tettük. Ezúttal azonban egyszerűen átmásolhatjuk a függvény definícióját a Delphi DLL forráskódjából:

```
function Double (N: Integer): Integer;
  stdcall; external 'FIRSTDLL.DLL';
function Triple (N: Integer): Integer;
  stdcall; external 'FIRSTDLL.DLL';
```

Ezek a deklarációk hasonlítanak azokhoz, amelyeket a C++ DLL hívásakor használtunk, ezúttal azonban nincs probléma a függvénynevekkel. A

változtatások ellenére a példa forráskódja ugyanaz marad, mint a CallCpp példánál, így ezt nem is tettem bele a könyvbe. A 20.5. ábra illusztrálja a DLL hívások hatását (a VCL alapú verzióban).



20.5. ábra *A Delphiben épített DLL-re hivatkozó CallFrst példaprogram kimenete*

Figyelem!

Ha a programot a DLL API alapú verzióját használva futtatjuk, akkor nem működik megfelelően, azaz a gombot többször is meg lehet nyomni, mielőtt bezárnánk az üzenetablakot. Ez azért van, mert a MessageBox API hívás első paramétere nulla, az értéke pedig a főform kezelője (handle) kellene lennie. Ennek a hibának a kijavítására írhatunk egy inicializáló függvényt, ami elmenti ezt az értéket a DLL adatai közé, és azt az értéket használjuk minden alkalommal, ha az egy másik függvényt hív meg. Az összetettebb Delphi DLL-ek esetében gyakran van szükség ilyen jellegű inicializáló függvényekre. Az ilyen esetekben mindig az a lényeg, hogy az alkalmazás globális objektumából a DLL megfelelő objektumába juttassunk át az értékeket. Más szóval a futtatható állomány Application objektumának a kezelőjét kell átmásolni a DLL Application objektumának ugyanezen tulajdonságába.

Delphi form a DLL-ben

Nem csak egyszerű, függvényeket és eljárásokat tartalmazó DLL-eket írhatunk, hanem akár egy teljes Delphiben készített formot is elhelyezhetünk DLL-ben. Ez lehet párbeszédablak vagy bármilyen másfajta form, és nem csak más Delphi programok használhatják, hanem más fejlesztői környezetek, sőt, makrónyelvek is.

Megjegyzés

Van két kivétel az általános szabály alól, miszerint a DLL-ek nem tartalmazhatnak Delphi objektumokat. Az egyik a csomagok, a másik pedig az ActiveX és az ActiveForm technológiák használata. Ezek a technológiák lehetővé teszik, hogy Delphi objektumot helyezzünk el egy bizonyos speciális DLL-ben, ami aztán tud működni OLE szerverként. Erről részletesen lesz szó a 21. és a 23. fejezetekben.

A FormDLL példa elkészítését egy már létező példa formját használtam fel (a ScrollC példáét a 8. fejezetből). Egyszerűen hozzávettem két bitképpel rendelkező gombot a formhoz, és átállítottam a BorderStyle (keretstílus) tulajdonságot `bsDialogra`. A form forráskódja, és típusdefiníciója semmi más változtatást nem tartalmaz, mindössze egy új függvényt tettem az egységbe, ami a formot definiálja. Az egység `interface` részét a következő sorral bővítettem:

```
function GetColor (Col: LongInt): LongInt; stdcall;
```

Az átadott paraméter lesz a kezdeti, a visszatérési érték pedig a végső szín (ami megegyezik a kezdeti színnel, ha a felhasználó a Cancel gombra kattint). Íme a függvény kódja, amit az egység implementációs részéhez adtunk hozzá:

```
function GetColor (Col: LongInt): LongInt;
begin
  // default value
  Result := Col;
  try
    FormScroll := TFormScroll.Create (Application);
    try
      with FormScroll do
      begin
        {initialize the data}
        Shape1.Brush.Color := Col;
        Shape2.Pen.Color := Col;
        ScrollBarRed.Position := GetRValue (Col);
```

```
            ScrollBarGreen.Position := GetGValue (Col);
            ScrollBarBlue.Position := GetBValue (Col);
            {show the form}
            if ShowModal = mrOK then
              Result := RGB (ScrollBarRed.Position,
                ScrollBarGreen.Position,
                ScrollBarBlue.Position);
          end; // with
        finally
          FormScroll.Free;
        end;
      except
        on E: Exception do
          MessageDlg ('Error in FormDLL: ' +
            E.MEssage, mtError, [mbOK], 0);
      end;
    end;
```

Ha Delphi komponenst (például formot) akarunk DLL-be helyezni, akkor csak olyan függvényeket használhatunk, amelyek létrehozzák, inicializálják, vagy futtatják a komponenst, illetve annak tulajdonságait, és adatait használják. A legegyszerűbb megközelítés az, ha veszünk egy függvényt ami beállítja az adatokat, lefuttatja a komponenst, majd visszaadja az eredményt, mint ebben a mostani példában. Bár komplexebb esetekben előfordulhat, hogy összetettebb adatstruktúrákat kell átadni paraméterként, amint azt tárgyaltuk is a „DLL paraméterek változatai" címszó alatt ebben a fejezetben.

Egy másik fontos elem a GetColor függvény kódjának szerkezete. A kód először létrehozza a formot, beállít bizonyos kezdeti értékeket, majd futtatja a megfelelő kódot, s ezáltal létrehozza a végső adatokat. Ezt a kódot mindössze az különbözteti meg az általában a programok belsejébe írt kódtól, hogy itt jelen van egy kivétel-kezelő programrész is.

Az egész függvényt védi egy `try-except` blokk, így bármilyen kivétel is keletkezik, a hiba nem okoz gondot, és a program megjeleníti a megfelelő hibaüzenetet. Azért kell minden lehetséges kivételt kezelni, mert a hívó eljárás bármilyen nyelven íródhatott, például olyanban is, amelyik nem tudja, hogy hogyan kell a felmerülő hibákkal foglalkozni. Néha akkor is hasznos lehet ezt a védő mechanizmust használni, ha a hívó egy Delphi program.

A formon végzett műveleteket a `try-finally` blokk védi. Ennek az a feladata, hogy a form objektum megfelelően megsemmisüljön, még akkor is, ha kivétel történik.

A ShowModal metódus visszatérési értékét megvizsgálva határozza meg a program a függvény eredményét. Beállítottam a kezdeti értéket a `try` blokk előtt, ezzel biztosítva, hogy a programrész mindig lefusson (és hogy a fordító ne figyelmeztessen, hogy a függvény esetleg nem kap értéket). Vegyük észre

azt is, hogy paraméterként átadtam egy hosszú egészet (`longint`), ami a Windows `COLORREF` adattípusához tartozik. A `TColor` Delphi típus használata lehet, hogy bonyodalmat okozott volna a nem Delphi alkalmazásoknál: bár a `TColor` nagyon hasonlít a `COLORREF`-hez, ezek a típusok nem teljesen azonosak. DLL írásakor ajánlatos kizárólag Windows-os adattípusokat használni (kivéve persze, ha a DLL-t kizárólag Delphi programok használják).

Most, hogy frissítettük a formot és megírtuk az egység kódját, áttérhetünk a projekt forráskódjára, ami a következő lesz:

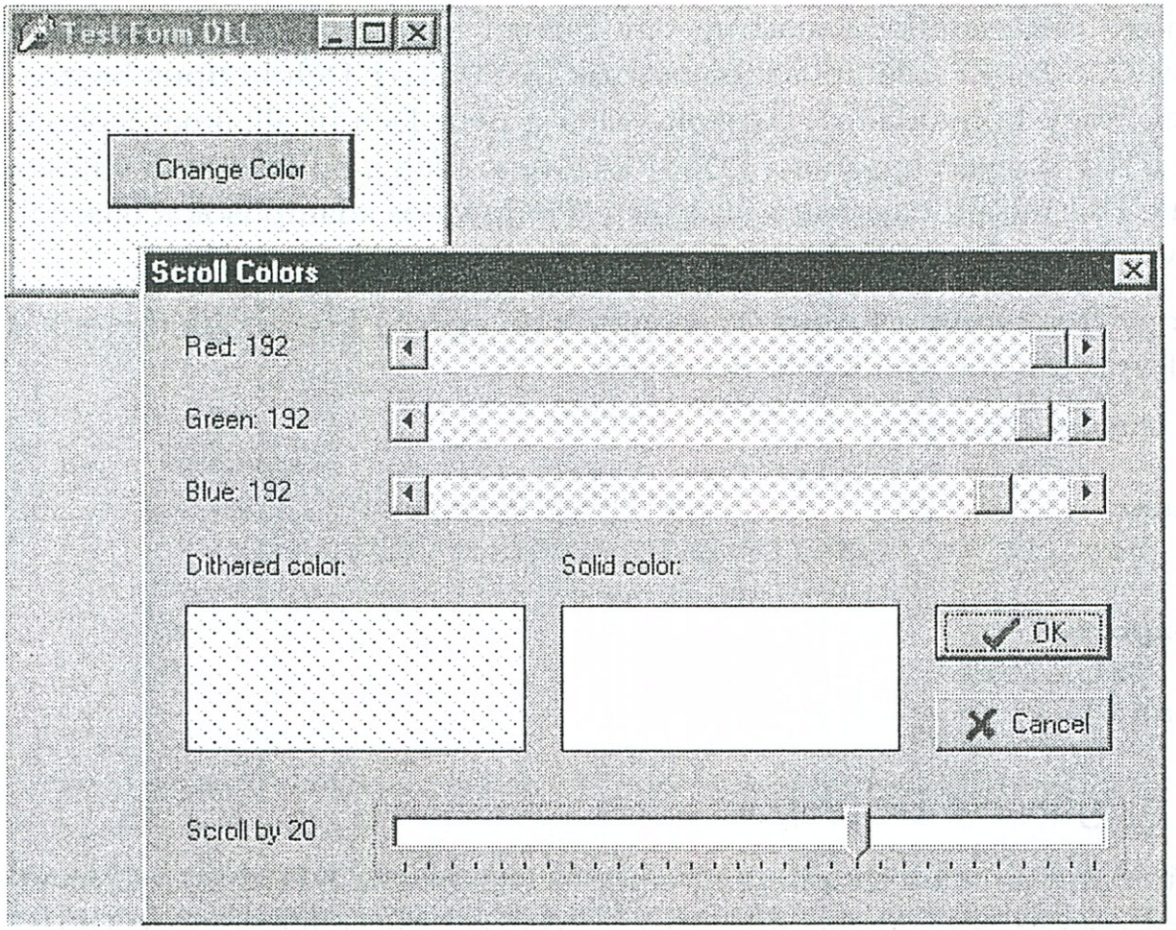
```
library FormDLL;

uses
  ScrollF in 'SCROLLF.PAS' {FormScroll};

exports
  GetColor;
end.
```

A DLL meghívása Delphiből

Most már kipróbálhatjuk Delphi programból a DLL-ben levő formot. A CD mellékleten a UseCol példa ugyanabban a könyvtárban van, mint az előző DLL (a FormDLL). A UseCol nevű példa igen egyszerű, formja csak egy gombot tartalmaz. Ha a felhasználó megnyomja a gombot, akkor lefut a DLL függvény. Itt következik ennek a függvénynek a definíciója és a Button1Click metódus kódja.

```
function GetColor (Col: LongInt): LongInt;
  stdcall; external 'FormDLL.DLL';

procedure TForm1.Button1Click(Sender: TObject);
var
  Col: LongInt;
begin
  Col := ColorToRGB (Color);
  Color := GetColor (Col)
end;
```

A program futtatásakor megjeleníti a párbeszédablakot az aktuális háttérszínnel (20.6. ábra). Ha megváltoztatjuk a színt és az OK-ra kattintunk, akkor a form háttérszíne a kiválasztott érték szerint változik.



20.6. ábra *A UseCol tesztprogram működése, amikor a FormDLL-ben levő párbeszédablakot hívja meg*

Delphi DLL hívása a Visual Basic for Applications nyelvből

Ezt a Color nevű párbeszédablakot más programozási nyelvekből is használhatjuk. C-ben vagy C++-ban egyszerű dolgunk van: csak annyit kell tenni, hogy létrehozunk egy import könyvtárat (az IMPLIB parancs segítségével), és a keletkező LIB fájlt hozzávesszük a projekthez. Mivel ebben a fejezetben már használtunk C++ fordítót, ezért most egy hasonló példát fogok mutatni, de a Microsoft Word for Windows-zal. Először is indítsuk el a Microsoft Wordöt, majd nyissuk ki a **Makró** párbeszédablakot (az Eszközök | Makró | Makrók menüponttal, vagy a megfelelő paranccsal attól függően, hogy melyik változatát használjuk a Wordnek). Ezután írjunk be egy makrónevet, például azt, hogy „DelphiColor", majd kattintsunk a Létrehozás gombra. Most már beírhatjuk az alábbi BASIC kódot, ami deklarálja a DLL-ünk függvényét, meghívja azt, majd az eredményt hozzáadja az aktuális dokumentumhoz (az Insert paranccsal), ami egyébként nem más, mint a szín jellemzése a vörös, zöld, kék arányokkal (RGB), majd pedig a végértéket megjeleníti a státuszsorban (a Print meghívásával):

```
Declare Function GetColor Lib "FormDLL"(Col As Long)
    As Long
Sub Main
  NewColor = GetColor(0)
  Print "The code of the color is " + Str$(NewColor)
  Insert "Red:" + Str$(NewColor Mod 256) + Chr$(13)
  Insert "Green:" + Str$(Int(NewColor / 256) Mod 256)
    + Chr$(13)
  Insert "Blue:" + Str$(Int(NewColor / (256 * 256)))
    + Chr$(13)
End Sub
```

Sajnos a Wordben nem éppen egyszerű RGB színeket használni, mivel a Word színsémái fix színkódokon alapulnak. Íme tehát egy lehetséges eredménye a fenti makrónak.

```
Red: 141
Green: 109
Blue: 179
```

Az a legszebb az egészben, hogy a három sort a fenti makró segítségével készítettem, amikor ezt a szöveget írtam. Ennek a makrónak a szövege megtalálható a WORDCALL.TXT nevű fájlban a hozzá tartozó DLL könyvtárában. Ha ki akarjuk próbálni, akkor ne felejtsük el a DLL fájlt bemásolni a PATH környezeti változóban megadott valamelyik mappába, vagy a windows\system mappába. Természetesen ennek a programnak a futtatásához szükség van a Microsoft Wordre. Más Microsoft Office alkalmazás (és valószínűleg más programok makrónyelvei is) a fentihez igen hasonló kódot várnak.

Megjegyzés

Egyszerűbben is építhetünk be Delphi kódot Office alkalmazásokba, ha ahelyett, hogy makrónyelvekből hívnánk meg őket, egyszerűen csak az OLE Automationt használjuk. Az OLE Automation-re majd a 22. fejezetben látunk példákat.

Ikonokat tartalmazó DLL készítése

A DLL-eknek számos egyéb felhasználási területük van a Windows-ban. Példaként most építünk egy ikonokat tartalmazó DLL-t, majd dinamikusan betöltjük az ikonokat egy programba. Most fogunk először alkalmazásból hivatkozni DLL-re futásidőben, ellentétben az eddigiekkel, amikor fordítási időben szerkesztettük be őket. Ezzel azt érjük el, hogy a program akkor is fut, ha a DLL nem elérhető.

Megjegyzés

Az ikonokról, erőforrásokról és erőforrás fájlokról a 27. fejezetben található bővebb információ. Ez a példa nagyon egyszerű, így bizonyára könnyű lesz követni.

Ahhoz, hogy ikonokból vagy más Windows-os erőforrásból építhessünk dinamikus könyvtárat, mindössze annyit kell tenni, hogy előkészítünk egy erőforrásfájlt és beszerkesztjük a könyvtárba. Példáként előkészítettem egy három erőforrást tartalmazó fájlt, a neveik Icon_A, Icon_B és Icon_C. Ha elkészült a fájl (ezúttal ICONS.RES néven), megírhatjuk a DLL forráskódot:

```
library Iconsdll;
{$R ICONS.RES}
uses
  Windows;
end.
```

A uses kulcsszóra csak azért van szükség, hogy a Delphi fordító ne panaszkodjon, hogy a projekt fájl szerkezete nem megfelelő. Fordítsuk le ezt a programot, és az ICONSDLL.DLL könyvtár már használható is. Tekinthetjük ezt a DLL-t ikonok és más erőforrások (mint például szabványos hibaüzenetek, vagy bittérképek), gyűjtőhelyének, amiket egyszerre több alkalmazás is használhat. S ha két alkalmazás használja ezt a DLL-t akkor az erőforrásoknak csak egy példánya kerül be a memóriába. Ha viszont ugyanazt az erőforrást több program is tartalmazza, akkor minden egyes program betölt egy-egy példányt a memóriába.

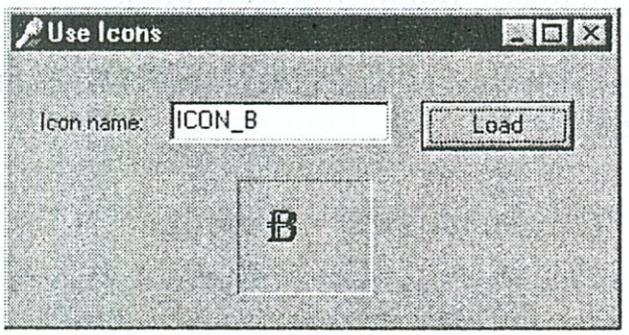
Ikonok betöltése DLL-ből

Most, hogy kész van a DLL, már csak be kéne tölteni egy programba. Erre a célra készítettem egy egyszerű példaprogramot amivel a felhasználó megadhatja annak az ikonnak nevét, amit látni szeretne (a program neve UseIcons, és az ICONSDLL könyvtárban van). Ennek az alkalmazásnak a formja tartalmaz egy szerkesztőmezőt a megfelelő címkével, egy gombot, és egy képet kerettel. Amikor a form létrejött, a program megpróbálja megnyitni a DLL-t, a LoadLibrary API függvény meghívásával:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  HInst := LoadLibrary ('Iconsdll.dll');
  if HInst = 0 then
    LoadButton.Enabled := False;
    ShowMessage ('Icons DLL not found');
  end;
end;
```

A LoadLibrary függvény visszaadja a DLL példány kezelőjét. Ha ez az érték 0, az azt jelzi, hogy hiba történt (a hiba pontos meghatározásához meg kell hívni a GetLastError függvényt). A DLL kezelője a form egy külön mezőjében kerül elmentésre (HInst) mert a későbbiekben szükség lesz rá a LoadButtonClick metódusban ahhoz, hogy a DLL-ből ikont tölthessünk be (vegyük észre, hogy a gombra csak akkor lehet kattintani, ha a könyvtár betöltése sikerült). Ennél az eseménykezelőnél az ikon név alapján töltődik be az Edit komponensben levő szöveg alapján. Ha a program már betöltette az ikont, akkor a kezelő az új kép megfelelő beállítására használható (20.7. ábra):

```
procedure TForm1.LoadButtonClick(Sender: TObject);
var
  HIcon: THandle;
begin
  HIcon := LoadIcon (HInst, PChar(Edit1.Text));
  if HIcon = 0 then
    ShowMessage ('Icon not found')
  else
    Image1.Picture.Icon.Handle := HIcon;
end;
```



20.7. ábra *A UseIcons program kimenete*

Természetesen az is elképzelhető, hogy a program nem találja meg az ikont. Ebben az esetben a felhasználó hibaüzenetet kap, és a kép változatlan marad. A program utolsó része is nagyon fontos. Ha manuálisan töltünk be egy könyvtárat a LoadLibrary függvény segítségével, akkor fel is kell szabadítani a FreeLibrary hívással:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FreeLibrary (HInst);
end;
```

Ha elfelejtjük felszabadítani a könyvtárat, akkor egészen addig bennmarad a memóriában, amíg ki nem lépünk a Windows-ból.

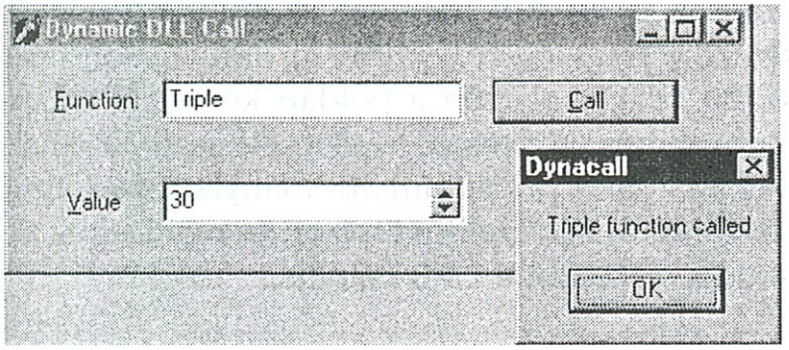
DLL függvény meghívása futásidőben

Most, hogy már tudjuk, hogyan lehet elérni a DLL-ben lévő erőforrásokat futásidőben, ugyanezzel a megközelítéssel megpróbálhatunk függvényeket is elérni. Erre készítettem egy nagyon egyszerű, könnyen átalakítható példát. Először bemutatom a programot, aztán ismertetek néhány általános esetet, amikor az általa használt megközelítés hasznos lehet. A példa neve DynaCall, és a fejezetben korábban ismertetett FirstDLL könyvtárat használja (hogy a program működőképes legyen, bemásoltam a DLL-t az új példa könyvtárába). Ebben a példában ahelyett, hogy deklarálnánk a Double és Triple függvényeket és egyszerűen használnánk őket, inkább egy valamelyest komplexebb kódot alkalmazunk. E módszer előnye az, hogy ily módon a program eléri a DLL új függvényeit is anélkül, hogy újrafordítanánk.

Ebben a példában a form csak egy gombot, egy szerkesztőmezőt és egy SpinEdit komponenst tartalmaz. Ha a gombra kattintunk, lefut a program egyetlen metódusa. Először a metódus meghívja a LoadLibrary függvényt, majd ha a könyvtár példányának a kezelője érvényes, akkor lefut a GetProcAddress API függvényt. Ez a függvény végignézi a DLL-t a paraméterként kapott függvénynév után kutatva. Ha a GetProcAddress megtalálja a függvénynevet, akkor visszaad egy, a kívánt eljárásra mutató pointert. Most egyszerűen csak ellátjuk ezt a függvénypointert a megfelelő adattípussal, és meghívjuk. E hívás hatása, és a program futásának eredménye a 20.8. ábrán látható. Itt pedig a (kicsit bonyolult) kód következik:

```
type
  TIntFunction = function (I: Integer): Integer;
stdcall;

procedure TForm1.Button1Click(Sender: TObject);
var
  HInst: THandle;
  FPointer: TFarProc;
  MyFunct: TIntFunction;
begin
  HInst := LoadLibrary ('Firstdll.dll');
  if HInst > 0 then
  try
    FPointer := GetProcAddress (HInst,
      PChar (Edit1.Text));
    if FPointer <> nil then
    begin
      MyFunct := TIntFunction (FPointer);
      SpinEdit1.Value := MyFunct (SpinEdit1.Value);
    end
    else
      ShowMessage ('DLL function not found');
```

```
    finally
      FreeLibrary (HInst);
    end
    else
      ShowMessage ('Library not found');
end;
```



20.8. ábra *A DynaCall program kimenete*

Hogyan kell meghívni egy Delphi függvényt, ha egyszer megvan a rá mutató pointer? Az egyik lehetséges megoldás az, ha konvertáljuk a mutatót eljárás típusúvá, majd meghívjuk az eljárást egy eljárás típusú változó segítségével. Ezt tettem a fenti programban. Ne felejtsük azonban el, hogy a definiált eljárástípusnak kompatíbilisnek kell lennie a rutin DLL-béli definíciójával. Ez a módszer Achilles-sarka: nincs mód a paramétertípusok ellenőrzésére.

Mi hát az előnye ennek a megközelítésnek? Elméletileg bármely DLL tetszőleges függvényét akármikor elérhetjük vele. A gyakorlatban akkor hasznos, ha különböző DLL-jeink vannak kompatíbilis függvényekkel, vagy egyetlen DLL több kompatíbilis függvénnyel, mint a mi esetünkben. Egyszerűen úgy is meghívhatunk egy függvényt, hogy beírjuk a nevét a szerkesztőmezőbe. Ha ezek után valaki ad nekünk egy DLL-t amelyben van egy új függvény, ami egész értéket vár bemenő adatként, és egész érték az eredménye is, akkor azt is meghívhatjuk úgy, ha beírjuk a nevét a szerkesztőmezőbe. Nem kell újra lefordítani az alkalmazást.

Ezzel a kóddal a fordító nem ellenőrzi, hogy a DLL létezik-e. Amikor a program elindul, a DLL nem töltődik be azonnal. A programot meg rugalmasabbá tehetjük, ha megadjuk a felhasználónak a lehetőséget, hogy ő adja meg a használandó DLL nevét. Bizonyos esetekben ez igen komoly előnyökkel járhat. A program futás közben is válthat DLL-t. Ezt a direkt módszer nem engedi meg. Másrészt a DLL függvények betöltésének ez a módja rendkívül elterjedt a makrónyelvekben, és a vizuális programozási környezetekben. A korábban ismertetett Word makró is ezzel a módszerrel él a DLL betöltése és a külső függvény hívása során, s ugye a Wordöt senkinek nincs kedve újrafordítani?

Csak egy fordítóra épülő nyelv (mint például a Delphi) engedheti meg magának a direkt hozzáférést, ami általában megbízhatóbb, stabilabb, és egy kicsit gyorsabb is. Szerintem a DynaLink indirekt töltési módszere csak speciális esetekben hasznos, ugyanakkor rendkívül hatékony eszköz is lehet.

A DLL a memóriában

Ezt (az egyébként a GetProcAddress nevű API függvényre épülő) technikát arra használhatjuk, hogy lekérdezzük, hogy egy függvény az aktuális folyamat melyik memóriacímére képeződött le. Ugyanezen a példán keresztül fogjuk megmutatni, hogy két Win32 program könnyedén elérheti egy DLL-nek ugyanazt a globális adatát, mivel az adat a privát címtartományban van. Ez a DllMem nevű példa két projekten alapul: egy Delphi DLL-en és egy Delphi alkalmazáson. Íme a teljes forráskódja a DLLMEM.DPR fájlnak:

```
library dllmem;

uses
  SysUtils;
var
  Data: Integer; // global DLL data

procedure SetData (I: Integer); stdcall;
begin
  if I <> Data then
    Data := I
  else
    Beep;
end;

function GetData: Integer; stdcall;
begin
  Result := Data;
end;

exports
  SetData, GetData;
begin
  // initialization
  Data := 1;
end.
```

A program két egyszerű rutint tartalmaz, ami egy globális memóriaterületen elhelyezkedő változót használ, aminek kezdeti értéke 1. Ha ugyanazt az értéket újra eltároljuk, akkor a könyvtár figyelmeztető hangjelzést ad.

Ezt a DLL-t használó program kódja igen egyszerű (a program neve UseMem). Az alkalmazás formja tartalmaz három gombot, egy címkét és két

szerkesztőmezőt, amelyek közül az egyikhez tartozik egy léptetőgomb is. Az első gomb menti el az első szerkesztőmezőben levő értéket, ami a hozzákapcsolt léptetőgombon keresztül kap értéket:

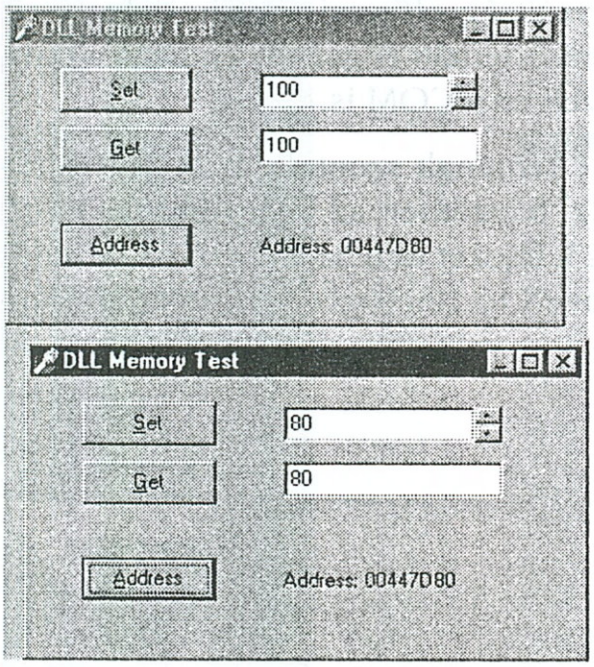
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SetData (UpDown1.Position);
end;
```

Ha megnyomjuk a második gombot, akkor a program bemásolja a DLL adatait a második szerkesztőmezőbe:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Edit2.Text := IntToStr(GetData);
end;
```

Végül a harmadik gomb megnyomásakor megjelenik a DLL SetData függvényének memóriacíme a címkén.

```
procedure TForm1.Button3Click(Sender: TObject);
var
  HDLLInst: THandle;
begin
  HDLLInst := LoadLibrary ('dllmem');
  Label1.Caption := Format ('Address: %p', [
    GetProcAddress (HDLLInst, 'SetData')]);
  FreeLibrary (HDLLInst);
end;
```



20.9. ábra *Ha a UseMem programból két példányt indítunk el, akkor észrevehetjük, hogy a USEMEM.DLL globális adatain nem osztoznak*

Ha a programot két példányban futtatjuk (a 20.9. ábra szerint), akkor láthatjuk, hogy minden példányban más az értéke a DLL globális adatának. Másrészt Windows 95-ben a program minden példánya ugyanazon a memóriacímen hivatkozik a függvényre, a Windows NT esetében viszont ezek az értékek eltérhetnek. Tulajdonképpen az a tény, hogy Windows 95 így viselkedik, nem bizonyítja, hogy a DLL csak egyszer töltődik be a memóriába, majd minden egyes alkalmazás címtartományában ugyanoda képeződik le, s nem többször töltődik be ugyanarra a címre. Egy globális elérhetőségű adat vélhetően ugyanazon a címen van minden folyamat címtartományában, de különböző fizikai memóriaterületekre képeződik le.

Hogyan tovább?

Ebben a fejezetben eddig azt tárgyaltuk, hogy hogyan kell C++-ban és más nyelveken írt DLL-ek függvényeit meghívni, és hogy hogyan lehet Delphiben DLL-t készíteni. A DLL-ek használata a hagyományos módszerek közé sorolható a különböző programozási nyelvekben és programozási környezetekben írt programok összeboronálására. A COM és az OLE ennél is fejlettebb módszereket tartalmaznak, de mivel ez az új technológia még mindig a DLL-ekre épül, ezért a következő fejezetet a DLL-en belüli osztályok használatával kezdjük.

Amikor a DLL és a többi alternatíva közül kell választanunk, ne felejtsük el, hogy a Delphinek és a Windowsnak sok közös eleme van, valamint hasonló a programozási „filozófiájuk" is. Hacsak lehet, kövessük a Delphi objektum-orientált szemléletét a Windows eljárás-orientált szemléletével összhangban, s nem fogjuk megbánni. Az objektum-orientáltság előnyei annyira jelentősek, hogy a Microsoft belevett néhány objektum-orientált fogalmat magába a rendszerbe is. Ilyen beépített technológia az OLE és a COM is.

A következő fejezeteket teljes egészében e témának szenteltem. A 21. fejezetben ismertetem a COM-ot és az OLE-t, a 22. fejezetben az OLE Automation-ről és az OLE dokumentumokról beszélek, a 23. fejezet témája az ActiveX vezérlők fejlesztése, a 24. fejezet pedig néhány ActiveX technológia internetes alkalmazását hivatott bemutatni.

21. FEJEZET

OLE és COM

E fejezet tartalma:

- Mi is voltaképpen az OLE és a COM?
  (Objektumok, osztályok, COM interfészek)
- A Shell interfész használata

OLE és COM

Amennyiben hinni lehet a Microsoftnak, akkor az OLE technológia nagy szerephez jut a jövőben a Windows alkalmazásokban. A Microsoft az OLE szót több különböző fogalomra is használja. Hogy ennek okát megértsük, ahhoz egy kicsit közelebbről szemügyre kell vennünk az OLE-t. Ezzel fogjuk kezdeni ezt a fejezetet. Ezután megvizsgáljuk az OLE alapját képező objektummodellt, (ezt hívják COM-nak). Ebben a fejezetben megpróbálom bemutatni, hogy az OLE valójában sokkal egyszerűbb, mint hinnénk. Az OLE egyébként azért tűnik bonyolultnak, mert kevés világos leírás született róla eleddig. Ebben a fejezetben egy egyszerű Delphi DLL-ből kiindulva fokozatosan felépítjük az első COM objektumunkat, ez remélhetőleg segíteni fog a technológia alapelemeinek elsajátításában. A fejezet vége felé mutatok majd néhány példát a Win32 API-ban levő OLE interfészek használatára is, az OLE tényleges alkalmazásairól az ezt követő fejezetekben lesz szó.

Mi is voltaképpen az OLE és a COM?

Az OLE körül kialakult értetlenség részben annak tudható be, hogy a Microsoft az OLE szót több értelemben is használja. Az egész az Object Linking and Embedding (röviden OLE) technológiával kezdődött, ami a DDE (Dynamic Data Exchange) modell egy kiterjesztése volt. A vágólap lehetővé teszi, hogy nyers adatot másoljunk egyik alkalmazásból a másikba, a DDE-vel két dokumentum bizonyos részeit tudjuk összekapcsolni, az OLE segítségével pedig a kiszolgáló alkalmazásból a kliens alkalmazásba juttathatunk át adatot, néhány, a kiszolgáló alkalmazásra vonatkozó információval (esetleg a Windows Registry-ben tárolt információra való hivatkozással) egyetemben. A nyers adat átvihető a kapcsolattal együtt (beágyazás), de az eredeti fájlban is bent tartható (csatolás). Ezt a technológiát a következő fejezetben tárgyaljuk részletesen.

Megjegyzés

Az OLE technológiával csatolt illetve beágyazott dokumentumokat nevezik OLE dokumentumoknak. Ezek egy újabb elnevezése az Active Document.

A Microsoft átírta az OLE-t, így született az OLE 2, és újabb lehetőségeket épített bele, mint például az OLE Automation és az OLE kontrolok használata. A következő lépésként megírták a Windows shellt az OLE technológiájával, és interfészével, majd pedig átnevezték az OLE kontrolokat ActiveX kontrolokká (ezeket azelőtt OCX-nek hívták), ezáltal megnyitva a kapukat az internetes terjesztés előtt.

Ahogy ez a technológia fejlődésnek indult, és egyre nagyobb jelentőségre tett szert, a Microsoft átnevezte OLE-nek. Ez már nem is rövidítés, hanem egyszerűen csak egy név volt. Ezek a névváltoztatások csak részben jelentenek tényleges technológia újításokat, sokkal inkább üzletpolitikai indokokkal magyarázhatóak.

Akkor, most mi az OLE? Alapvetően az OLE egy technológia, ami azt definiálja, hogy a szerver modul, és a kliens modul hogyan kommunikál egymással egy adott interfészen keresztül. Itt most a modul szó alkalmazásokat, illetve DLL-eket jelent, amelyek futhatnak egyazon, de különböző gépeken is. Sokféle interfész közül lehet válogatni a kiszolgáló és a kliens program feladatától függően, valamint új interfészt is létre lehet hozni esetleges speciális célokra. Ezeket az interfészeket a kiszolgáló objektumok (szerverobjektumok) tartalmazzák. Egy szerverobjektum általában több interfészt is meghatároz, és minden szerverobjektumnak vannak közös részei, hiszen mindegyiknek tartalmaznia kell például az IUnknown interfészt. Most pedig nézzük át az OLE által használt objektummodellt (ez a Component Object Model).

Definíció szerint a COM az OLE implementációja. A gyakorlatban azonban a két szót összevissza használják egymás helyett. A COM definiálja az OLE technikai részleteit, és minden COM kompatibilis nyelv alkalmas COM/OLE objektumok írására. Eredetileg C és a C++ nyelveken íródtak ezek a kódok, de a Delphi teljes mértékben megfelel a COM szabványnak. Ha vetünk egy pillantást a forráskódokra, akkor feltűnik, hogy Object Pascalban könnyebb COM objektumokat írni, mint C++-ban. Ez a viszonylagos egyszerűség többnyire a Delphi 3-ban bevezetett interfészeknek tudható be. Egyébként ugyanez az interfészes megközelítés tette lehetővé, hogy a Javat integrálják a COM-mal a Windows-os felületeken.

Megjegyzés

Az OLE és a COM teljes leírását áhítóknak Kraig Brockschmidt Inside OLE című könyvének második kiadását ajánlom (a könyv a Microsoft Press gondozásában jelent meg). Ez a könyv teljes, de nem éppen egyszerű leírása az OLE-nek. Vannak benne C, és C++ példák, de nem tartalmaz Delphi kódot.

Az OLE tehát interfészek halmaza. Érdemes megjegyezni, hogy ezeknek a moduloknak a feladata két szoftver modul, két futtatható fájl, illetve egy futtatható fájl és egy DLL közötti kommunikáció megvalósítása. Általában azért egyszerűbb DLL-ekkel megvalósítani bizonyos objektumokat, mert Windows 95 alatt

a program és az általa használt DLL egy címtartományban vannak. Ez azt is jelenti, hogy ha a program átad egy memóriacímet a DLL-nek, akkor a cím érvényes marad. Két futtatható fájl esetében viszont az OLE-ra komoly feladatok hárulnak a kommunikáció megvalósításában. Ezt a mechanizmust *marshalling*-nak hívják.

Jegyezzük meg, hogy az OLE objektumokat megvalósító DLL-t *in-process server* (folyamaton belüli szerver) névvel illetik, míg ha a szerver egy különálló futtatható állomány, akkor *out-of-process* (folyamaton kívüli) szerverként hivatkoznak rá. Ebben a fejezetben többnyire in-process szerverekkel fogunk foglalkozni, amik tulajdonképpen DLL-ben lévő OLE szerverek. A következő fejezetekben szó esik majd az out-of-process szerverekről is. Mindenekelőtt azonban tisztázni kell, hogy hogyan tudunk Delphiben DLL objektumokat exportálni.

Objektumok a DLL-ekben

Amint azt az előző fejezetekben láttuk, a DLL-ek tulajdonképpen függvénygyűjtemények. Azonban azt is megtehetjük, hogy definiálunk egy objektumot egy DLL-ben, majd egy programból használjuk az osztály bizonyos függvényeit. Valójában a DLL általában nem exportál osztályt, csak az osztály objektumainak virtuális metódusait, más szóval a DLL-nek kell létrehoznia az osztály objektumait. Ha a DLL létrehozott egy objektumot, akkor visszaadhatja azt a hívó alkalmazásnak. Minden objektum tartalmaz egy pointert, ami annak virtuális metódusaira mutat, ezért amikor a DLL visszaadja az objektumot, akkor automatikusan visszaadja az objektum osztályához tartozó virtuális metódusok címét is (lásd a késői kötésről – late binding – szóló részt az 5. fejezetben). Természetesen a futtatható fájlnak fel kell ismernie ezeket az objektumokat, és tartalmaznia kell egy ugyanolyan osztálydeklarációt, mint amilyen a DLL-ben van.

Osztályok egy DLL-en belül

A következő példaprogramban (DllObj) bemutatom, hogy hogyan lehet DLL-ben osztályt definiálni, és exportálni annak virtuális metódusait. A DLL projekt tartalmaz egy olyan egységet ami az exportálandó absztrakt osztály definícióját tartalmazza, valamint egy másikat, ami az implementációs osztály definícióját. Íme az első egységben levő Base nevű absztrakt osztály definícióját tartalmazó egység:

```
type
  TDllNumber = class
  public
    function GetValue: Integer;virtual;abstract;
    procedure SetValue (New: Integer);virtual;abstract;
    procedure Increase; virtual; abstract;
  end;
```

Megjegyzés

Az ilyen osztályokat, amelyekben csak virtuális és absztrakt metódusok vannak, gyakran tisztán absztrakt osztályoknak nevezzük. Amint rövidesen látni fogjuk, ez a struktúra tulajdonképpen igen közel áll az interfészhez.

A `TDllNumber` osztály annyira egyszerű, hogy szinte teljesen használhatatlan. Ám most sokkal inkább a benne használt technikát szeretném ismertetni, mintsem magát az osztályt. Ebből a deklarációból kitűnik, hogy minden metódusnak virtuálisnak, és absztraktnak kell lennie. Csak a virtuális metódusok kerülnek be a VTable-be (a virtuális metódusok táblázatába), és szintén csak ők exportálhatóak a DLL-ből. Az implementációs osztály ebből az absztrakt osztályból származik, és a DLL egy külön egységében található:

```
type
  TDllNumberImpl = class (TDllNumber)
  private
    Value: Integer;
  public
    constructor Create;
    function GetValue: Integer; override;
    procedure SetValue (New: Integer); override;
    procedure Increase; override;
  end;
```

Az osztály tényleges kódja semmi különöset nem tartalmaz:

```
constructor TDllNumberImpl.Create;
begin
  Value := 0;
  inherited Create;
end;
function TDllNumberImpl.GetValue: Integer;
begin
  Result := Value;
end;
procedure TDllNumberImpl.SetValue (New: Integer);
begin
  Value := New;
end;
procedure TDllNumberImpl.Increase;
begin
  Inc (Value);
end;
```

A most bemutatott osztályon kívül a DLL második egységének tartalmaznia kell egy olyan függvényt, ami létrehozza az osztály objektumait.

```
function NewObject: TDllNumber;
begin
  Result := TDllNumberImpl.Create;
end;
```

Ez a függvény is igen egyszerű. A DLL projekt fájl exports részébe kell beleírni, és `stdcall`-ként kell deklarálni:

```
library DllObj;
uses
  DllClass in 'DllClass.pas';
exports
  NewObject;
end.
```

Ennyiből áll tehát Delphi osztályt exportálni DLL-ből. Az alaposztályt akár ki is hagyhattuk volna, és írhattunk volna egyetlen osztályt aminek aztán minden metódusa virtuális lett volna, de a fentiekben használt megközelítés (külön absztrakt, és virtuális osztály definiálása) ennél sokkal elegánsabb, és csak egy kicsivel igényel több odafigyelést. Ez így még nem egy COM kompatibilis osztály, de a most következő példákban azzá tesszük.

Osztály használata DLL-ből

Most pedig megpróbáljuk felhasználni ezt az osztályt egy programból, létrehozva néhány objektumot. Az első teendő a programban, hogy újradeklaráljuk az osztályt. Mivel az absztrakt osztály deklarációja egy külön egységben van, ezért a programban elég egyszerűen arra az egységre hivatkozni. Ezúttal nem fogjuk megírni az implementációs osztályt, hanem a DLL implementációs osztályára hagyatkozunk. Tulajdonképpen az alaposztály definícióiban lévő virtuális és absztrakt metódusok csak arra valók, hogy hibátlan legyen a DLL osztályának VTable-je a DLL-t használó programban is.

Valójában a kliensprogramnak elég, ha van egy olyan interfésze, melynek virtuális metódusai ugyanolyan paraméterekkel rendelkeznek, és ugyanabban a sorrendben is vannak felsorolva.

```
type
  TDllNumber = class
  public
    function GetValue: Integer; virtual; abstract;
    procedure SetValue (New: Integer);virtual;abstract;
    procedure Inc; virtual; abstract;
  end;
```

A változás itt csak annyi, hogy a metódus nevét átírtam Increase-ről Inc-re. Ez egyáltalán nincs hatással a program működésére.

Figyelem!

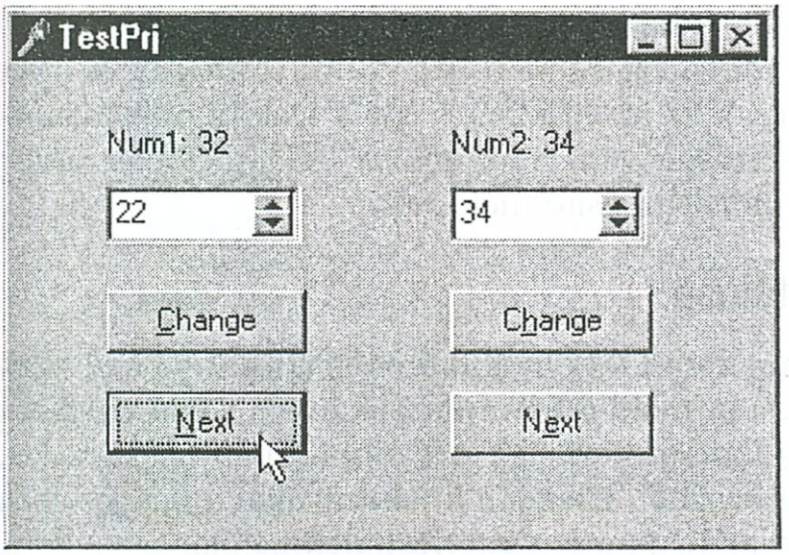
Azt, hogy a metódus neve miért nincs hatással a program működésére elég bonyolult megmagyarázni. Van két deklaráció, egy a DLL-ben, egy pedig a programban. Ebben a két deklarációban a virtuális metódusoknak csak a sorrendje és a paraméterezése számít, nem pedig a neve. Amikor a Delphi program meghív egy virtuális függvényt, akkor tulajdonképpen azt a függvényt hívja meg, amelynek a címe a VTable adott pozíciójában található. Így tehát ha például ez a harmadik virtuális metódus az osztály lokális példányában, akkor a DLL által definiált osztály VTable-jének a harmadik pozíciójához fog tartozni, akármi is legyen a neve. Ezért létfontosságú, hogy a két definícióban ugyanazt a sorrendet kövessük. Ha átírjuk a definíciók sorrendjét, azaz például az Inc-et a SetValue elé írjuk, akkor a futó program a VTable második metódusát fogja nézni, és hibásan működik.

A Base egységben levő TDllNumber osztálydeklarációt használó tesztprogram igen egyszerű. A 21.1. ábrán látható a formja futásidőben. Amikor a form létrejön, a program készít két objektumot a TDllNumber osztályból, úgy, hogy meghívja a DLL függvényt, beolvassa az objektumok kezdeti értékeit a két SpinEdit komponensből, majd két címkén megjeleníti az objektumok aktuális értékeit.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // first create the object
  Num1 := NewObject;
  Num1.SetValue (SpinEdit1.Value);
  Label1.Caption := 'Num1: ' +
       IntToStr (Num1.GetValue);

  // create second object
  Num2 := NewObject;
  Num2.SetValue (SpinEdit2.Value);
  Label2.Caption := 'Num2: ' +
       IntToStr (Num2.GetValue);
end;
```

A DLL függvényt a szokásos módon definiáljuk:

```
function NewObject: TDllNumber; stdcall;
    external 'DllObj.dll';
```



21.1. ábra *A DllObj könyvtárat tesztelő TestPrj példaprogram kimenete egy DLL, ami egy osztály metódusait exportálja*

Ez a deklaráció nagyon fontos, hiszen ez biztosítja, hogy a DLL a program indulásakor automatikusan betöltődjön. E nélkül a futtatható fájl és a DLL nem is tudna kommunikálni. Itt következik az első objektumhoz tartozó két gomb kódja:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // change
  Num1.SetValue (SpinEdit1.Value);
  Label1.Caption := 'Num1: ' + IntToStr(Num1.GetValue);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Num1.Increase;
  Label1.Caption := 'Num1: ' + IntToStr(Num1.GetValue);
end;
```

Amint a fenti példa mutatja, elég egyszer megadni a DLL-nek, hogy hozza létre az objektumot, és utána bármikor használhatjuk ugyanúgy, mint bármely másik Delphi objektumot. A különbség csak annyi, hogy a metódusok nem a futtatható állomány, hanem a DLL részét képezik. Van még egy eseménykezelő a tesztprogramban, ennek feladata az objektum megsemmisítése.

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Num1.Free;
  Num2.Free;
end;
```

A FormDestroy metódusban meghívhatjuk bármely objektumnak a Free metódusát, ez pedig szokás szerint meghívja minden objektum Destroy nevű megsemmisítő metódusát. Ebben csak az az érdekes, hogy a Delphi objektumoknak eggyel több virtuális metódusuk van, mint amit a deklarációban adunk nekik, és ez az egy a Destroy megsemmisítő metódus.

Az IUnknown implementálása

Ahhoz, hogy a sima objektumunkat COM objektummá alakítsuk fel kell használni az IUnknown interfészt, ami majd a három szabványos COM metódust szolgáltatja (Add, Release és a QueryInterface). A Delphi 3 támogatja ezeket, csak új interfész kulcsszavakat használ. Ezeket a kulcsszavakat a 6. fejezetben ismertettük. Következzék most az IUnknown interfész definíciója:

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj):
        Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

Az AddRef és a Release metódusok feladata a hivatkozások számlálása. A QueryInterface metódus kezeli az objektum típusaira vonatkozó információkat, és vizsgálja az egyes típusok kompatíbilitását.

Megjegyzés

A fenti kódban láthatunk példát out típusú paraméterre. Ezt a paramétert a metódus adja vissza a hívó programnak, de úgy, hogy a hívó program nem ad neki kezdeti értéket.

IUnknown metódusok implementálása

Tulajdonképpen a Delphi tartalmazza ezeket a metódusokat némelyik osztályban. Például a `TInterfacedObject`-ben a következőképpen valósul meg az interfészt:

```
type
  TInterfacedObject = class(TObject, IUnknown)
  private
    FRefCount: Integer;
```

```
  protected
    function QueryInterface(const IID: TGUID; out Obj):
        Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    property RefCount: Integer read FRefCount;
  end;
```

A hivatkozásszámlálásra használt két igen egyszerű metódus megvalósítása pedig így néz ki:

```
function TInterfacedObject._AddRef: Integer;
begin
  Inc(FRefCount);
  Result := FRefCount;
end;

function TInterfacedObject._Release: Integer;
begin
  Dec(FRefCount);
  if FRefCount = 0 then
  begin
    Destroy;
    Result := 0;
    Exit;
  end;
  Result := FRefCount;
end;
```

A legbonyolultabb metódus a QueryInterface, ami a Delphiben a TObject osztály GetInterface metódusán keresztül valósul meg:

```
function TInterfacedObject.QueryInterface(
    const IID: TGUID; out Obj): Integer;
const
  E_NOINTERFACE = $80004002;
begin
  if GetInterface(IID, Obj) then
    Result := 0
  else
    Result := E_NOINTERFACE;
end;
```

A QueryInterface szerepe kettős. Egyrészt típusellenőrzést kell végeznie, hogy a program megkérdezhesse egy objektumtól, hogy „Te olyan típusú vagy amilyen nekem kell?", vagy „Te tartalmazod azt a metódust és interfészt amit én meg akarok hívni?" Ha a válasz nemleges, akkor a program nekiállhat újabb objektum után nézni, esetleg egy másik szerveren. Ha a válasz igen akkor a QueryInterface az objektumra mutató pointert adja vissza.

Ahhoz, hogy a QueryInterface metódus szerepét megérthessük fontos, hogy tisztában legyünk azzal a ténnyel, hogy egyetlen COM objektum több interfészt is létrehozhat. A QueryInterface meghívásakor a TGUID paraméter segítségével hivatkozhatunk egy adott interfészre a sok közül.

Általában egy COM osztály több interfészt is tartalmazhat, így előfordulhat, hogy a VTable egy bizonyos részét vissza kell adni. Delphi 2-ben ezt körülményes volt megvalósítani, míg C++-ban egyszerűen megoldható a többszörös öröklődés segítségével. A Delphi 3 igen elegáns módon oldja meg a problémát. Egy osztály több interfészt is implementálhat.

Globális egyedi azonosítók

A QueryInterface metódusnak van egy speciális TGUID típusú paramétere. Ez az azonosító azonosítja a rendszer minden COM szerver osztályát és interfészét. Ha arra vagyunk kíváncsiak, hogy egy objektum támogat-e egy adott interfészt, akkor mindössze meg kell kérdeznünk tőle, hogy tartalmazza e annak azonosítóját. (Az OLE interfészek azonosítóját a Microsoft határozza meg.)

Egy adott osztály illetve egy adott szerver jelölésére egy másik azonosítót használnak. Ez utóbbi azonosító a Windows Registry-ben található, a hozzá kapcsolódó DLL illetve futtatható fájl megjelölésével együtt. Egy-egy osztály azonosítóját az OLE szerver fejlesztői határozzák meg.

A fent említett két azonosító mindegyikét GUID-ként emlegetik (Global Unique Identifier, magyarul globálisan egyedi azonosítók). Viszont ha minden fejlesztő a maga kis azonosítóját használja az egyes OLE szerverek a jelölésére, akkor hogyan lehetünk biztosak abban, hogy két egymástól független OLE szerver azonosítója különböző? A kérdésre az egyszerű válasz az, hogy sehogy. A hivatalos válasz persze az, hogy ezek a számok olyan nagyok (128 biten tárolt, azaz körülbelül 38 jegyű decimális számok), hogy statisztikailag lehetetlen, hogy két véletlen számnak ugyanaz legyen az értéke. Tanácsos továbbá meghívni a CoCreateGuid API-t (vagy használni az adott programozási környezetben levő megfelelő opciót), ami olyan azonosítót hoz létre, amelyben szerepel a dátum, a pontos idő, a hálózati kártya azonosítója, a hívó program, és más egyéb információk.

A Delphi (a System egységben) definiál egy TGUID típust, ami ezeket a számokat tartalmazza:

```
type
  TGUID = record
    D1: Integer;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

Ez a struktúra ugyan elég furcsa, de ezt követeli meg a Windows. A GUID-hoz a szabványos hexadecimális jelöléssel is rendelhetünk értéket, amint azt a következő példa illusztrálja:

```
const
  Class_ActiveForm1: TGUID =
    '{1AFA6D61-7B89-11D0-98D0-444553540000}';
```

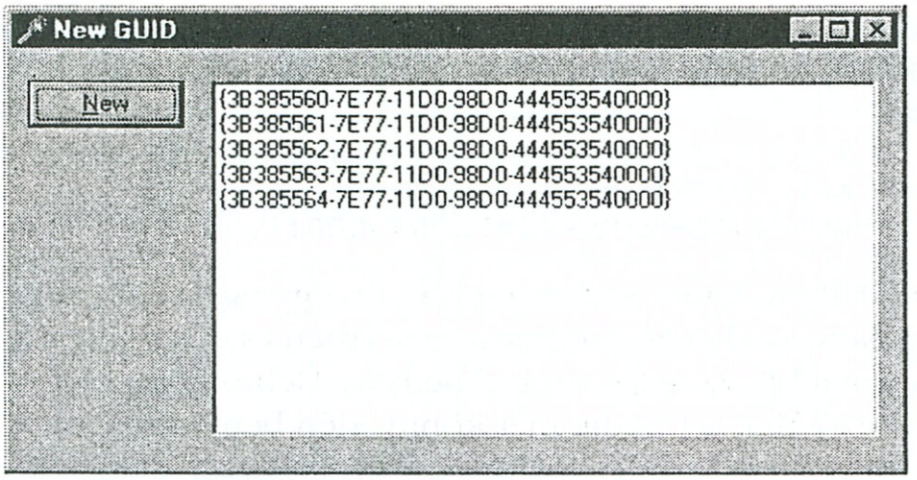
Arra az esetre, ha netán GUID-re lenne szükségünk írtam egy egyszerű programot a generálásukra. A NewGuid példaprogram egyszerűen csak meghívja a CoCreateGuid függvényt, majd hozzáadja az eredményt a Memo komponens szövegéhez. Az így keletkező Memo-t pedig már könnyedén bemásolhatjuk a kódba.

Figyelem!

Amellett, hogy nagyon ügyelünk arra, hogy nehogy lemásoljuk a GUID-et másvalaki programjából (aminek az lenne a vége, hogy van két teljesen különböző COM objektum ugyanazzal a GUID-del), ne tegyük azt sem, hogy hasunkra ütve beírunk egy számot ID gyanánt. A Windows ellenőrzi az ID-ket, és a találomra kiválasztott szám nem lesz érvényes GUID. Az érvénytelen GUID-dal rendelkező OLE szervert egyrészt nem ismeri fel a rendszer, másrészt még csak hibaüzenetet sem küld.

Íme az OnClick eseménykezelő formjában levő egyetlen gomb kódja (a 21.2. ábra mutatja a kód működését):

```
uses
  ComObj, ActiveX;

procedure TForm1.BtnNewClick(Sender: TObject);
var
  ID: TGUID;
  S: string;
begin
  if CoCreateGuid (Id) = s_OK then
  begin
    s := GUIDToString (Id);
    Memo1.Lines.Add (S);
  end;
end;
```



21.2. ábra *Egy példa a NewGuid példaprogrammal generált GUID-ra. Az értékek a számítógéptől, valamint a futtatás idejétől függnek*

Megjegyzés

Az OLE függvényeket és adattípusokat definiáló Delphi egységet OLE2-nek nevezik Delphi 2 alatt. A Delphi 3-ban ezt az egységet az új interfész mechanizmussal működő ActiveX és ComObj egységek helyettesítik. Az új verzióban ajánlatos csak ezeket az egységeket használni.

A GUIDToString függvényen kívül a párja is megtalálható a Delphiben. Ez a StringToGUID függvény. A megfelelő Windows API függvényt is használhatjuk (StringFromGuid2), de akkor a string típus helyett a WideString típust kell elővenni. Valahányszor OLE-t használunk, mindig WideString típust kell használni, kivéve ha Delphi függvényekkel dolgozunk, amelyek automatikusan elvégzik a szükséges konverziókat. Valójában az OLE API függvények a PWChar típust használják (ami nem más mint egy olyan pointer, ami bővített karakterek – wide character – nullával lezárt vektorára mutat), ám WideStringet illesztve a PWChar-ra áthidalható ez a probléma.

Megjegyzés

Ne felejtsük el, hogy a GUID-eknek két fajtája van: az Interfész ID, ami az interfész-re vonatkozik, és a Class ID, ami egy adott szerver egy bizonyos objektumát azonosítja. Minkét azonosítófajta GUID stílusú, és gyakran GUID-ként emlegetik mindegyiket. A két fajta közötti eltérés oly csekély, hogy az elnevezések felcserélhetők.

Ha megtekintjük a 21.2. ábrán, vagy akár élőben a program kimenetét, akkor láthatjuk, hogy az alkalmazás folyamatosan kapja a GUID-eket. S mindez teljesen időfüggetlen: akár lassan vagy gyorsan nyomogatjuk a gombot, a számok sorban egymás után generálódnak. Ám ha újabb példányt indítunk a programból, akkor az már másik értéksorozattal fut.

Az osztálygyárak szerepe

Az első olyan program, amelyben osztályt exportáló DLL-t használtunk azért működött, mert zárt környezetben dolgoztunk, azaz mi írtuk meg a szervert, és az egyetlen klienst is. Ahhoz, hogy ez szabványos COM objektummá váljon, használni kell az IUnknown interfészt, definiálni a GUID-ot, majd kicserélni a NewObject függvényt a DllGetClassObject függvényre. Amikor az objektum GUID-jét regisztráljuk a nyilvántartásban, akkor használhatunk API függvényt is az objektum létrehozásához. Ilyen például a CreateComObject API:

```
function CreateComObject(const ClassID: TGUID):IUnknown;
```

Ez az API függvény megnézi a nyilvántartást, megkeresi az adott GUID-hez tartozó objektum szerver regisztrációját, betölti, majd pedig ha a szerver DLL, akkor meghívja a DLLGetClassObject metódust a DLL-ből. Ezt a függvényt minden in-process szervernek tartalmaznia, és exportálnia kell:

```
function DllGetClassObject(const CLSID, IID: TGUID;
  var Obj): HResult; stdcall;
```

Ez az API függvény paraméterként kapja meg a kért osztályt és interfészt, és visszaad egy objektumot. A függvény által visszaadott objektumot hívjuk osztálygyárnak (class factory).

Akkor tehát mi az osztálygyár? Amint azt a neve is sugallja, ez egy olyan objektum, ami képes más objektumokat létrehozni. Egy szervernek több objektuma lehet. A szerver beindítja az osztálygyárat, az osztálygyár pedig létrehoz egy objektumot. Egy objektumnak lehet több interfésze. A Delphi 3 leegyszerűsített COM filozófiájának egyik előnye éppen az, hogy a rendszer tud osztálygyárakat létrehozni.

A CreateComObject API nem fejezi be futását az osztálygyár létrehozása után. Ha az osztálygyár létrejött, a CreateComObject meghívja a CreateInstance metódust az IClassFactory interfészből. Ez a metódus létrehozza a kívánt metódust, majd visszaadja. Ha ezalatt nem történik hiba, akkor ez az objektum lesz a CreateComObject API visszatérési értéke.

Azzal, hogy ezt a mechanizmust felépítettük, immár nagyon egyszerűen hozhatunk létre új objektumokat a CreateComObject függvényhívással, bár a hátterében igen komplex mechanizmus húzódik meg. A Delphi 3-ban az a jó, hogy ezt a komplex mechanizmust a futásidejű rendszer kezeli helyettünk. Elérkezett hát a pillanat, hogy részleteiben megvizsgáljuk, hogy a Delphi 3 miként egyszerűsíti le számunkra a COM használatát.

Delphi 3 COM interfészek használata

Most már foghatjuk a példaprogramunkat, és átírhatjuk úgy, hogy ugyanazt tudja mint a DllObj példaprogram, csak éppen a szabványos Delphi 3-as megközelítésben tegye mindezt. Íme a könyvtár által exportált osztály definíciója (bár itt már nem osztálynak, hanem interfésznek hívjuk):

```
type
  INumber = interface
    ['{B4131140-7C2F-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    procedure Increase; stdcall;
  end;
```

Ismét arra inteném az olvasót, hogy ne másolja be ezt az IID-t (Interfész ID) a kódjába, hanem inkább írjon egy másikat. A testre szabott interfész deklarálása után a szerver deklarál egy tényleges objektumot is, ami tartalmazza az interfészt:

```
type
  TDllNumber = class (TComObject, INumber)
  private
    Value: Integer;
  public
    function GetValue: Integer; virtual;stdcall;
    procedure SetValue (New: Integer); virtual;stdcall;
    procedure Increase; virtual;stdcall;
    procedure Initialize; override;
    destructor Destroy; override;
  end;
```

A TComObject osztály

A `TDllNumber` osztály örököl a szabványos `TComObject` osztálytól. Erről a Delphi 3-beli ComObj egység gondoskodik, és implementálja az alapértelmezett COM objektumot. Ez az osztály hasonlít a korábbiakban tárgyalt `TInterfacedObject` osztályhoz, csak éppen egy kicsit összetettebb. Itt következik a `TComObject` osztály deklarációjának publikus része:

```
type
  TComObject = class(TObject,IUnknown,ISupportErrorInfo)
    ...
  public
    constructor Create;
    constructor CreateAggregated(const Controller:
        IUnknown);
    constructor CreateFromFactory(Factory:
        TComObjectFactory; const Controller: IUnknown);
```

```
    destructor Destroy; override;
    procedure Initialize; virtual;
    function ObjAddRef: Integer; virtual; stdcall;
    function ObjQueryInterface(const IID: TGUID;
        out Obj): Integer; virtual; stdcall;
    function ObjRelease: Integer; virtual; stdcall;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): HResult; override;
    property Controller: IUnknown read GetController;
    property Factory: TComObjectFactory read FFactory;
    property RefCount: Integer read FRefCount;
  end;
```

Ez az osztály tartalmazza az IUnknown interfészt (az ObjAddRef, a ObjQueryInterface, és az ObjRelease metódusokat), valamint az ISupportErrorInfo interfészt (az InterfaceSupportErrorInfo védett metóduson keresztül). Tartalmaz néhány csak olvasható tulajdonságot is, amelyeket beolvasva információkat nyerhetünk az osztályokról valamint az objektum státuszáról. A példaprogramban egyszerűen csak öröklünk ettől az alaposztálytól, de nem törődünk túl sokat annak megvalósításával, csupán megjegyezzük, hogy ő lát el bennünket az alapvető COM funkciókkal.

További Delphi COM osztályok

A mostani példában használt `TComObject` osztályon és a 6. fejezetben használt `TInterfacedObject` osztályon kívül vannak még további előre definiált COM osztályok is a Delphi 3-ban. A következő fejezetekben használni is fogjuk őket. Mindenestre itt van egy lista a Delphi VCL legfontosabb COM osztályairól (ezek hierarchikus ábrázolása is megtalálható a 7. fejezetben):

- A System egységben definiált `TInterfacedObject` örököl a `TObject`-től, és tartalmazza az IUnknown interfészt.
- A ComObj egységben definiált `TComObject` örököl a `TObject`-től és tartalmazza az IUnknown és az ISupportErrorInfo interfészeket.
- A ComObj egységben definiált `TTypedComObject` örököl a `TComObject`től, és (a `TComObject` alaposztályban már meglévő IUnknown és ISupportErrorInfo interfészeken kívül) tartalmazza az IProvideClassInfo interfészt is.

További Delphi COM osztályok (folytatás)

- A GomObj egységben definiált `TAutoObject` örököl a `TTypedComObject`-től, és tartalmazza az IDispatch interfészt.
- Az AxCtrls egységben definiált `TActiveXControl` örököl a `TAutoObject`-től és számos interfészt tartalmaz (IPersistStreamInit, IPersistStorage, IOleObject, és IOleControl, hogy csak néhányat említsünk).

A Delphi ezen osztályok mindegyikére definiál osztálygyárat. Az osztálygyár osztályok is egy ugyanilyen hierarchiát képeznek. A neveik `TComObjectFactory`, `TTypedComObjectFactory`, `TAutoObjectFactory`, és `TActiveXControlFactory`.

COM objektumok inicializálása

Egy valamit azonban mindenképpen szeretném felhívni a figyelmet. A `TComObject` osztály nem virtuális konstruktora meghívja a virtuális Initialize metódust. Ha a saját ízlésünk szerint szeretnénk objektumot létrehozni, akkor se definiáljunk új konstruktort (ami mellesleg sohasem kap hívást). Ehelyett inkább írjuk felül az Initialize metódust, amint azt a `TDllNumber` osztálynál meg is tettem fentebb. Továbbá az osztály megsemmisítőjét is felülírtam, mert le akartam tesztelni a Delphi automatikus COM objektum megsemmisítő szolgáltatását. Íme az álkonstruktor, és a megsemmisítő kódja:

```
procedure TDllNumber.Initialize;
begin
  inherited;
  Value := 10;
end;

destructor TDllNumber.Destroy;
begin
  inherited;
  MessageBox (0, 'Object Destroyed',
    'TDLLNumber', mb_OK); // API call
end;
```

Az első metódus, amelyik az örökölt verziót hívja meg jó gyakorlat, még akkor is, ha a TComObject.Initialize metódusnak nincs is kódja ebben a Delphi verzióban. A destruktornak meg kell hívnia az alaposztályt. Ez a kód kell tehát ahhoz, hogy a COM objektum megfelelően működjön. A másik három metódus (az INumber interfész metódusai) továbbra is ugyanaz mint az előző két példában, így azok itt nem is szerepelnek.

Még akad néhány teendő mielőtt a példaprogramunk elkészül. Először is azonosítót kell adnunk a szerver objektumnak, lehetőség szerint konstanst használva:

```
const
  CLSID_TDllNumber: TGUID =
    '{95D47840-7C4D-11D0-98D0-444553540000}';
```

Osztálygyárat is kell készítenünk. A Delphi 3 tartalmaz egy „kulcsrakész" osztálygyárat a `TComObject`-ből származó osztályokhoz. Ezt a `TComObjectFactory` névvel illetik. Ez az osztály tartalmazza mind az IUnknown mind pedig az IClassFactory interfészeket, valamint exportál rengeteg tulajdonságot is, amint az a publikus interfészéből ki is tűnik:

```
type
  TComObjectFactory = class(TObject,
    IUnknown, IClassFactory)
  public
    constructor Create(ComServer: TComServerObject;
      ComClass: TComClass; const ClassID: TGUID;
      const ClassName, Description: string;
      Instancing: TClassInstancing);
    destructor Destroy; override;
    function CreateComObject(
      const Controller: IUnknown): TComObject; virtual;
    procedure RegisterClassObject;
    procedure UpdateRegistry(
      Register: Boolean); virtual;
    property ClassID: TGUID read FClassID;
    property ClassName: string read FClassName;
    property ComClass: TClass read FComClass;
    property ComServer: TComServerObject
      read FComServer;
    property Description: string read FDescription;
    property ErrorIID: TGUID read FErrorIID
      write FErrorIID;
    property ProgID: string read GetProgID;
    property Instancing: TClassInstancing
      read FInstancing;
  end;
```

Most mindössze annyit kell tennünk, hogy a COM kompatibilis osztályainkat definiáló egységben globális objektumot készítünk ebből az osztályból. Ezt az egység `initialization` részében tehetjük meg:

```
initialization
  TComObjectFactory.Create( ComServer, TDllNumber,
    CLSID_TDllNumber, 'Md3.FirstCom',
    'Md3 First COM Server', cimultiInstance);
end.
```

A fenti kód objektumot készít a osztálygyárból, paraméterként átad egy ComServer objektumot, átadja továbbá a most definiált osztály referenciáját (TDllNumber), a most definiált GUID-ot, a szerver nevét, és a szerver jellemzését.

A ComServ egységben definiált globális ComServer objektum az adott egységben elérhető osztálygyárat adminisztrálja. A saját ForEachFactory metódusával keresi ki az adott COM objektum kérését támogató osztályt. Másrészt pedig nyomon követi a létrehozott objektumok számát.

Két lépést kell még megtennünk ahhoz, hogy a szerverünk teljes mértékben COM kompatibilis legyen. Meg kell írni a megfelelő interfész függvényeket, és a regisztrációs mechanizmust. Ez elég egyszerű, hiszen a ComServ egység tartalmazza ezeket a függvényeket, így a szerverünknek egyszerűen csak exportálnia kell őket. Itt következik a FirstCom szerver példaprogram projekt forráskódja:

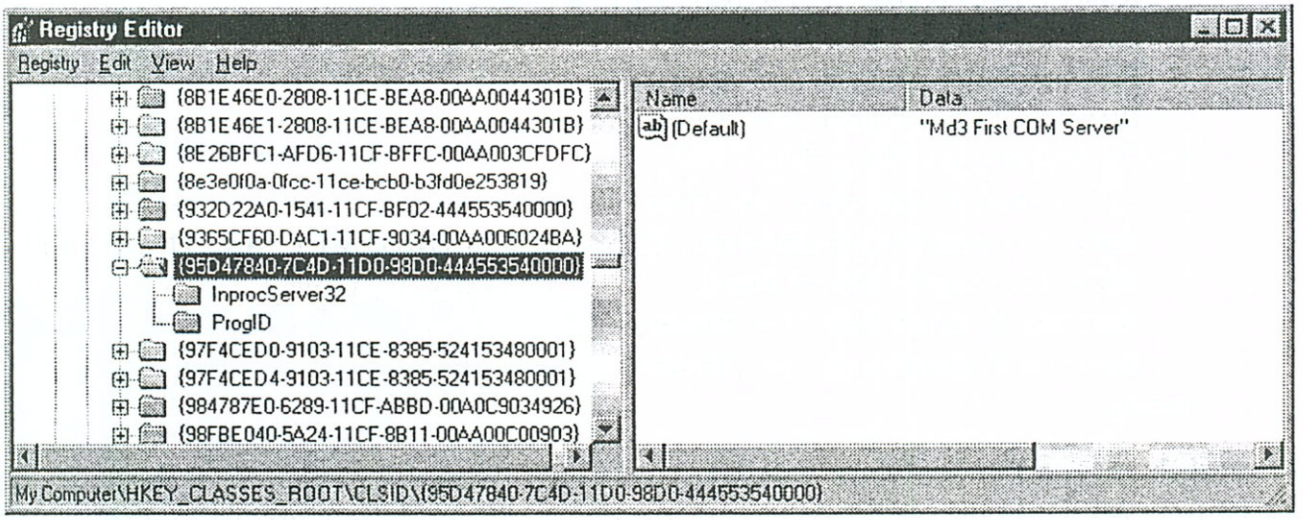
```
library FirstCom;

uses
  ComServ,
  DllClass in 'DllClass.pas';

exports
  DllGetClassObject resident,
  DllCanUnloadNow resident,
  DllRegisterServer resident,
  DllUnregisterServer resident;
end.
```

Megjegyzés

A DLL export rezidens direktíváinak használata nem más, mint form optimalizáció. Ezek a függvények mindig bent maradnak a memóriában, így gyorsan meghívhatóak, még akkor is, ha már régen nem használtuk őket (bár lehet, hogy ekkor a Windows már a merevlemezen lévő virtuális memóriába helyezte őket).

A szerver regisztrálásához elég egyszerűen csak lefordítani a kódot, majd kiválasztani a **Run** menüből **Register ActiveX Server** parancsot a Delphi környezetben. Ezzel a saját gépünkön regisztráltuk a szervert a 21.3. ábrán látható eredménnyel. Ha ezt a szervert eljuttatjuk a felhasználókhoz, akkor a kliensgépeken telepíteni kell. Ezt többféleképpen is megtehetjük: írhatunk .REG fájlt,

hogy automatikusan telepítődjön a Registry-ben, átadhatjuk a DLL-t parancssor paraméterként a Microsoft REGSVR32.EXE-nek (ez utóbbi a Windows System könyvtárában van), de használhatjuk a Delphi részét képező RegSvr nevű demóprogramot is. Most már nézhetjük a példánkat a kliens oldalról. Ezúttal a példa neve TestCOM és külön könyvtárban van. Ezzel is ki kívánom hangsúlyozni, hogy ezúttal a program tölti be a szerver DLL-t az OLE/COM mechanizmuson keresztül a nyilvántartásban szereplő információ segítségével. Így a kliensnek nem kell tudnia, hogy a szerver melyik könyvtárban van.



21.3. ábra *Az újonnan regisztrált Windows95 szerver adatai a Registryben*

A COM kliens program frissítése

A kliens programban deklarálnom kell az interfészt és a COM szerver GUID-t (bár tehettem volna egy olyan közös egységbe őket, amit a szerver és a kliens is használ, ám úgy döntöttem, hogy nem végzem el ezt a plusz munkát, egyszerű példáról lévén szó):

```
const
  CLSID_TDllNumber: TGUID =
    '{95D47840-7C4D-11D0-98D0-444553540000}';

type
  INumber = interface
    ['{B4131140-7C2F-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    procedure Increase; stdcall;
  end;
```

Természetesen most a példaprogramban a FormCreate metódus kódját is frissíteni kell. A program indulásakor egyik gomb sem érvényes (amíg a kirajzolás történik). Csak akkor válnak elérhetővé, ha létrejön egy objektum. Így aztán ha valamelyik objektum létrehozása során kivétel lép fel, akkor az adott objektumhoz tartozó gomb nem lesz engedélyezett:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // create first object
  Num1:=CreateComObject (CLSID_TDllNumber) as INumber;
  Num1.SetValue (SpinEdit1.Value);
  Label1.Caption:='Num1: ' + IntToStr (Num1.GetValue);
  Button1.Enabled := True;
  Button2.Enabled := True;

  // create second object
  Num2:=CreateComObject (CLSID_TDllNumber) as INumber;
  Label2.Caption:='Num2: ' + IntToStr (Num2.GetValue);
  Button3.Enabled := True;
  Button4.Enabled := True;
end;
```

Figyeljük meg a CreateComObject hívását, és az azt követő as kulcsszót. Az API hívás elindítja az előzőekben részletezett objektumkészítő mechanizmust. Ez a hívás aztán dinamikusan be is tölti a szerver DLL-t. A visszatérési érték pedig egy IUnknown objektum. Mielőtt hozzárendelnénk ezt az objektumot a Num1 és a Num2 mezőkhöz át kell alakítani a megfelelő interfész típussá. Ezúttal a megfelelő típus az INumber.

Figyelem!

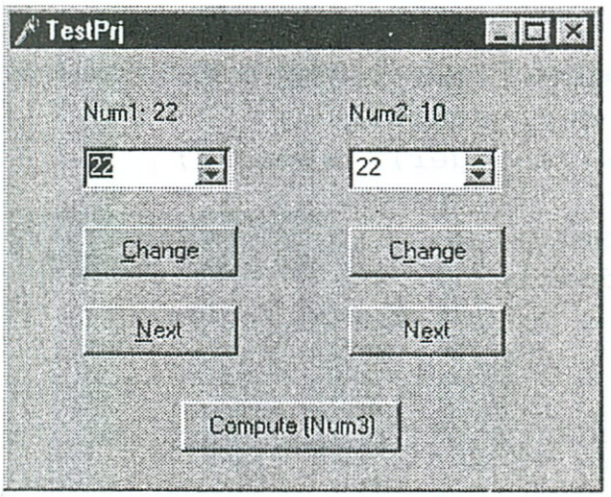
Ahhoz, hogy az interfészt megfeleltessünk az adott típusnak, mindig használjuk az as utasítást (amint a 6. fejezetben tárgyaltuk). Ez nyújt némi védelmet is, mivel kivétel keletkezik, ha az adott objektum nem támogatja az interfészünket. Másrészt az interfészekben az as használata az egyetlen módja annak, hogy objektumból interfészt vegyünk ki. Ha simán azt írjuk, hogy:

```
INumber(CreateComObject (CLSID_TDllNumber))
```

akkor a program egészen biztosan le fog fagyni, pedig a kód jónak tűnik. Az a probléma, hogy interfész pointer másik interfész pointerre alakítása hibának számít, úgyhogy soha ne használjuk ezt a technikát.

A 21.4. ábrán látható ennek a tesztprogramnak a kimenete. Szembetűnő a hasonlóság az előző változattal, vegyük azonban észre, hogy a Num2 ezúttal az objektum alapértelmezett értékét mutatja az Initialize metódusban tett beállításoknak megfelelően! Figyeljük meg azt is, hogy a formot új gombbal bővítettem, ami egy harmadik ideiglenes COM objektumot hoz létre.

```
procedure TForm1.Button5Click(Sender: TObject);
var
  Num3: INumber;
begin
  // create a new temporary COM object
  Num3:=CreateComObject (CLSID_TDllNumber) as INumber;
  Num3.SetValue (100);
  Num3.Increase;
  ShowMessage ('Num3: ' + IntToStr (Num3.GetValue));
end;
```



21.4. ábra *A TestCom példa kimenete. Szembetűnő hasonlóságot mutat az előző példa kimenetével, ám ezúttal száz százalékig COM-kompatibilis filozófiával a háttérben*

Ha ezt a gombot nyomjuk meg, akkor egyszerűen a 100 után következő számot kapjuk meg. Tulajdonképpen nem a kimenet miatt vettem hozzá ezt a metódust a példához (hiszen ugye mindenki tudja, hogy száz meg egy az százegy). Ha az eredményt jelző üzenet megjelenése után még egyszer megnyomjuk a gombot, akkor azt az üzenetet kapjuk, hogy az objektum megsemmisült. Ez azért van, mert ha egyszerűen csak eltűnik a láthatárról az objektum, akkor automatikusan lefut a Release metódus, az objektum referenciaszámlálója csökken eggyel, végül eléri a nullát, az objektum pedig megsemmisül. Ezt a referenciaszámlálót használó mechanizmust már ismertettem, tehát a további részletekre itt most nem térek ki. Ugyanez történik a többi objektummal is

mihelyt a program befejezi futását. A FormDestroy metóduson keresztül akkor is megsemmisülnek, ha a program explicit módon nem teszi ezt meg velük. Erről tanúskodik a Destroy destruktor által kiírt üzenet is. Mindez azért van így, mert interfész típusúként deklaráltuk őket.

Az interfész tulajdonságok használata

Egy újabb apró lépésben továbbfejleszthetjük a példaprogramot úgy, hogy tulajdonságot rendelünk az INumber interfész-hez. Amikor tulajdonságot rendelünk egy interfészhez, akkor jelöljük az adattípust, majd pedig a `read` és a `write` direktívákat. Lehetnek csak olvasható, illetve csak írható tulajdonságok is, de a `read` és a `write` rész mindig metódusra kell, hogy vonatkozzon, ugyanis az interfészek csak metódusokat tartalmazhatnak.

Íme a továbbfejlesztett interfész, ami egyébként a PropCom példa részét képezi:

```
type
  INumberProp = interface
    ['{B36C5800-8E59-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    property Value: Integer
      read GetValue write SetValue;
    procedure Increase; stdcall;
  end;
```

Ennek az interfésznek új nevet adtam, s ami még fontosabb új interfész azonosítót is, majd pedig egyszerűen átírtam a szerver osztálydeklarációit a következőképpen:

```
type
  TDllNumber = class (TComObject, INumberProp)
  ...
```

Ennek az osztálynak az objektumazonosítója is új. A TestProp könyvtárban levő kliensprogram most simán használhatja a Value tulajdonságot a SetValue és GetValue metódusok használata helyett. Következzék most egy részlet a FormCreate metódusból:

```
Num1 := CreateComObject (CLSID_TDllNumberProp) as
    INumberProp;
Num1.Value := SpinEdit1.Value;
Label1.Caption := 'Num1: ' + IntToStr (Num1.Value);
```

Az interfész esetében a metódusok és az interfészek használata között csak szintaktikailag van különbség, hiszen az interfész tulajdonságai nem férhetnek hozzá privát adathoz. A tulajdonságok használatával azonban egy kicsit olvashatóbbá tehetjük a kódot.

Megjegyzés

Vektor (array) tulajdonságokat is definiálhatunk, sőt ki is választhatunk egy adott vektortulajdonságot, hogy az legyen az interfész alapértelmezett vektortulajdonsága, ám e témákat a könyvben nem tárgyaljuk.

Shell interfész használata

Az utolsó részben építettünk egy teljesen szabványos COM objektumot, folyamaton belüli (in-process) szerverként package-et készítettünk belőle, majd pedig szabványos kliensből használtuk. Igaz, hogy a létrehozott COM interfész egy általunk épített számunkra testre szabott interfész volt. Most már megpróbálhatunk a Windows 95-ös shell interfészhez kapcsolódó klienseket, és szervereket készíteni, amelyek persze egytől egyig COM-ok.

Megjegyzés

Mindössze két egyszerű példát mutatok be, mert ha teljes mélységében próbálnánk összefoglalni a Windows 95 shellt, arra bizony rámenne néhány fejezet (egész könyvek foglalkoznak a témával.) Többek között a Delphi Developer's Handbook című könyvben található részletes leírás a Windows 95 shell interfészeiről.

Shell csatolások (vagy parancsikonok) készítése

Az egyik legegyszerűbb shell interfész, amit kliens alkalmazásban használhatunk, az IShellLink interfész. Ez az interfész szoros kapcsolatban áll a Windows 95 parancsikonokkal, és lehetővé teszi, hogy a programozó információt nyerjen egy létező parancsikonról, illetve hogy újat hozzon létre.

Az ShCut példában létrehozunk egy olyan parancsikont, ami magára a programra mutat. Persze ha az olvasó egyszer megérti, hogy hogyan is megy a dolog, akkor már könnyedén továbbfejlesztheti a programot. A példa tartalmaz egy szerkesztőmezőt és egy gombot. Ha megnyomjuk a gombot, akkor a szerkesztőmező tartalma lesz az új parancsikon:

```
uses
  ComObj, ActiveX, ShlObj;

procedure TForm1.Button1Click(Sender: TObject);
var
  AnObj: IUnknown;
  ShLink: IShellLink;
  PFile: IPersistFile;
  FileName: string;
  WFileName: WideString;
begin
  // access the two interfaces of the object
  AnObj := CreateComObject (CLSID_ShellLink);
  ShLink := AnObj as IShellLink;
  PFile := AnObj as IPersistFile;
  // get the name of the application file
  FileName := ParamStr (0);
  // set the link properties
  ShLink.SetPath (PChar (FileName));
  ShLink.SetWorkingDirectory (
      PChar (ExtractFilePath (FileName)));
  // save the file, using a WideString!
  WFileName := ExtractFilePath (FileName) +
      Edit1.Text + '.lnk';
  PFile.Save (PWChar (WFileName), False);
end;
```

A kód legfontosabb része a metódus legelején található. A CreateComObject hívás létrehoz egy rendszerobjektumot a paraméterként kapott GUID alapján. A hívás eredményét (ami egyébként egy IUnknown interfész) pedig átalakítja IShellLink interfésszé és IPersistFile-á is. Tulajdonképpen a kódot az alábbi parancsikon formátummal is meg lehetett volna írni:

```
ShLink:=CreateComObject (CLSID_ShellLink) as IShellLink;
PFile:=ShLink as IPersistFile;
```

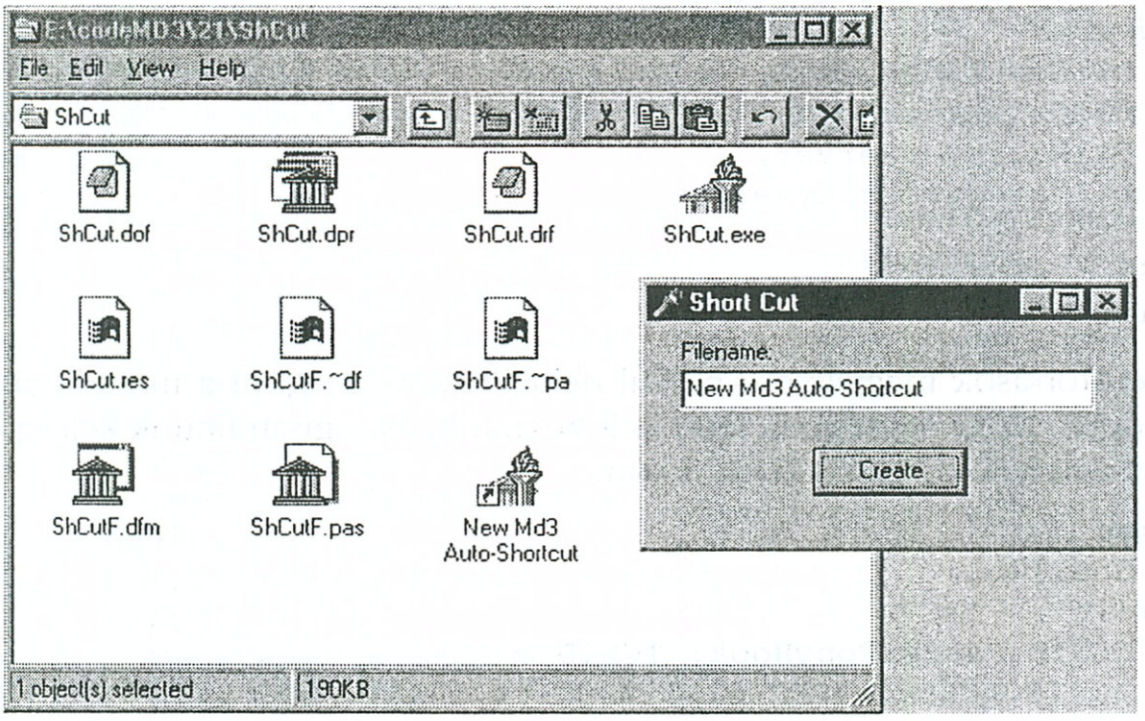
Ha az olvasó megtekint néhány Delphi 2-ben, vagy más nyelveken írt efféle példát, akkor észreveheti, hogy az IPersistFile interfész eléréséhez a programok a QueryInterface metódust hívják meg. Tulajdonképpen a két as kifejezés a QueryInterface-t használja.

Ha adott két interfész, amelyek mindegyike ugyanahhoz a szerverobjektumhoz kapcsolódik, akkor meghívhatjuk bizonyos metódusaikat (például SetPath, SetDirectory, Save). Ez az utolsó hívás (ami létrehozza a fizikai .LNK fájlt) egy bővített karakterre mutató mutató (pointer to wide char) típusú paramétert vár. A Delphi 3-ban ezt úgy lehet a legegyszerűbben megvalósítani, hogy deklarálunk egy long stringet, majd ráillesztünk egy PWChart.

Figyelem!

Ne próbáljunk sima stringet PWCharra illeszteni. A fordító nem fog ugyan panaszkodni, ám a program sem fog működni.

Ha futtatjuk ezt a programot, és megnyomjuk a gombot, akkor a projekt könyvtárában a Windows 95 létrehoz egy új linket. A program működésére példát is láthatunk a 21.5. ábrán.



21.5. ábra *A ShCut példa felhasználói felülete, valamint néhány vele készített parancsikon a projekt mappában*

Megerősítést kérő ablak Pascal fájlokhoz

Nézzünk most egy újabb példát arra, hogy hogyan lehet egy programot beleintegrálni a shellbe. Írjunk egy egyszerű szervert! Ismét úgy döntöttem, hogy egy egyszerű példával kezdem. Ezúttal a program egy megerősítő ablak, ami tulajdonképpen egy olyan szerver, ami az ICopyHook interfészt tartalmazza, és regisztrálja a rendszerben. Van már a rendszerben egy ilyen regisztrált szerver, ami minden alkalommal működésbe lép, valahányszor mappát másolunk,

mozgatunk vagy törlünk. A program feladata, hogy rákérdezzen, tényleg el kívánja-e végezni a műveletet. Nem lép működésbe viszont akkor, amikor ugyanezeket a műveleteket fájlokon végezzük el. A példaprogram neve egyébként PasCHook.

Új szervert építeni úgy tudunk, hogy egyszerűen kiválasztjuk a Delphi **File** menüjéből a **New** parancsot, és az **ActiveX** lapon kiválasztjuk az ActiveX Library-t. Ezzel kódot generáltunk egy üres COM szerverhez. Ez az előzőekben használt projekt forráskód COM szerveres megfelelője.

```
library Project1;

uses
  ComServ,
  Unit1 in 'Unit1.pas';
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}
end.
```

Most az erőforrások megjelölése nélkül nekiállhatunk megírni a másodlagos egység kódját. Az új egységben létre kell hoznunk, és regisztrálnunk kell egy új osztályt amelynek saját GUID-ja is van:

```
uses
  ShellObj;
const
  CLSID_PascalCopyHook: TGUID =
    '{80A06FA0-7DF2-11D0-98D0-444553540000}';
type
  TPasCopyHook = class(TComObject, ICopyHook)
  public
    function CopyCallback (Hwnd: THandle;
      wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar;
      dwSrcAttribs: DWORD; pszDestFile:
      PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
  end;
```

Az ICopyHook interfész, amit implementálni akarunk mindössze egy metódussal rendelkezik, és a következőképpen van definiálva:

```
type
  ICopyHookA = interface(IUnknown)
    function CopyCallback(Wnd: HWND;
      wFunc, wFlags: UINT;
```

```
        pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
        pszDestFile: PAnsiChar;
        dwDestAttribs: DWORD): UINT; stdcall;
    end;
    ICopyHook = ICopyHookA;
```

Ez az interfész annyiban különbözik a többi shell interfésztől, hogy közvetlenül használja fel a forrás-, és a célmappa nevét. A shell egyéb kiterjesztései e helyett egy meghatározott interfészen keresztül kapnak memóriablokkot, és fájl információt. Ha megpróbáljuk megvalósítani őket, akkor készüljünk fel arra, hogy sokkal több kódra lesz szükség, mint ennél a példánál.

Itt van az általam készített CopyCallBack másolás megerősítés (copy hook) metódus kódja. Ez a metódus `id_Yes`-t ad vissza, ha engedélyezi az adott mappaműveletet, `id_No`-t, ha az adott mappaműveletet nem szabad befejezni, és `id_Cancel`-t ha éppen több mappával dolgozunk, és meg akarjuk szakítani a műveletet. Ez utóbbi esetben a metódus minden mappánál lefut egyszer. Itt következik a kód, ami a művelet típusától függően szolgáltat valamelyest eltérő kimeneteket:

```
function TPasCopyHook.CopyCallback (Hwnd: THandle;
  wFunc, wFlags: UINT; pszSrcFile: PAnsiChar;
  dwSrcAttribs: DWORD;
  pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT; stdcall;
var
  Msg: string;
  MessType: TMsgDlgType;
begin
  Application.Handle := Hwnd;
  Msg := '';
  MessType := mtConfirmation;
  case wFunc of
   FO_COPY: Msg := Format (
     'Are you sure you want to copy the %s ' +
     'folder to the %s destination?',
     [pszSrcFile, pszDestFile]);
   FO_DELETE:
   begin
     Format ('Caution... the folder %s is ' +
       'about to be deleted, unless you say no.',
       [pszSrcFile]);
     MessType := mtWarning;
   end;
   FO_MOVE: Format (
     'Are you sure you want to move the %s ' +
     'folder to the %s destination', [pszSrcFile,
     pszDestFile]);
```

```
      FO_RENAME: Format (
        'Are you sure you want to rename the %s ' +
        'folder as %s', [pszSrcFile, pszDestFile]);
    end;
    if Msg <> '' then
      // ask for confirmation
      Result := MessageDlg (Msg, MessType,
                            mbYesNoCancel, 0)
    else
      Result := id_Yes;
  end;

  initialization
    TComObjectFactory.Create(ComServer, TPasCopyHook,
      CLSID_PascalCopyHook, 'PascalCopyHook',
      'CopyHook Demo from Mastering Delphi 3',
      cimultiInstance);
  end.
```

A kód elejénél lévő Application.Handle := HWnd teszi lehetővé, hogy a MessageDlg függvényt megfelelően használhassuk. Tulajdonképpen a CopyCallBack metódus paraméterként kapta meg a párbeszédpanelhez használandó ablak kezelőjét. Ha a MessageBox Windows API-t használjuk, akkor a kezelő első paraméterként megy át. Ha viszont VCL függvényt használunk, akkor ezt a paramétert az Application objektum Handle tulajdonságának a beállítására kell használni.

Ha egyszer megírtuk és lefordítottuk a programot, akkor szembe kell néznünk a legbonyolultabb lépéssel, az alkalmazás regisztrálásával. Ezúttal .REG fájlt kell írni ahhoz, hogy a DLL-t beleintegráljuk a shellbe. Regisztrálhatunk shell kiterjesztést egy adott fájlhoz is, de minden fájlhoz is. A mappákra vonatkozó shell kiterjesztéseket a Registry-ben a Directory bejegyzéssel jelölik. E bejegyzés alá kell beírnunk egy új kulcsot a CopyHookHandlers-hez, értékként pedig a szerverünk GUID-jét kell hogy adjuk neki. Természetesen a szerver CLSID-jét is regisztráltatni kell (ezt az első néhány utasítás intézi el). Íme a .REG fájl:

```
REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\ ↵
  {80A06FA0-7DF2-11D0-98D0-444553540000}]
@="PascalCopyHook"
[HKEY_CLASSES_ROOT\CLSID\ ↵
  {80A06FA0-7DF2-11D0-98D0-444553540000}\InprocServer32]
@="e:\\codeMD3\\21\\PasCHook\\PasCHook.dll"
"ThreadingModel"="Apartment"

[HKEY_CLASSES_ROOT\Directory\shellex\CopyHookHandlers]
@="PCopyHook"
```

```
[HKEY_CLASSES_ROOT\Directory\shellex\CopyHookHandlers\↵
PCopyHook]
@="{80A06FA0-7DF2-11D0-98D0-444553540000}"
```

Fontos, hogy némelyik hosszabb sort ketté kellett törjem, hogy beleférjen a nyomtatott oldalba a kód! (Ezen sorok végére ↵ karaktert raktam a fenti listában.) Ezt azonban nem tehetjük meg az igazi .REG fájlban. A regisztrációs fájl szögletes zárójelbe zárt folyamatos, egysoros szövegeket vár.

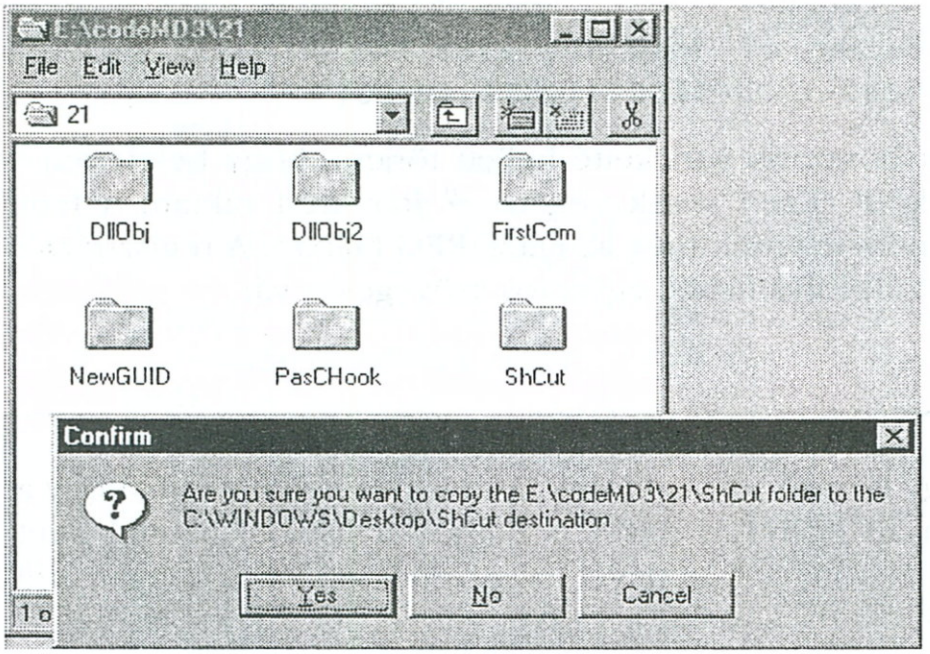
Figyelem!

Amellett, hogy hosszú sorokat írunk, arra is oda kell figyelni, hogy a DLL-t tartalmazó könyvtár nevét is beírjuk a listába. Ezúttal bent hagytam a saját könyvtáram nevét, hogy lehessen látni, hogyan kell használni a dupla backslash-t a szerver elérési útjának jelölésére.

A .REG fájlok használatának alternatívájaként az is megtehető, hogy a megfelelő API-val frissítjük a Windows Registry-t. A Delphinek erre egy külön osztálya van, a `TRegistry`. Ennek az osztálynak a használatával majd a 25. fejezetben ismerkedünk meg.

Ha egyszer már megfelelően lefordítottuk a programot, és telepítettük regisztrációs fájlját a Windows Registry-be – általában elég egyszerűen lefuttatni a .REG fájlt, és minden el van intézve – újra kell indítani a rendszert (bár vannak esetek, amikor ez az utóbbi lépés kihagyható). Mostantól ha bármikor el akarjuk végezni a fenti mappaműveletek valamelyikét (mozgatás, törlés, másolás, átnevezés), akkor a művelet végrehajtása előtt rá fog kérdezni a rendszer, hogy biztosan akarjuk-e. A kimenetre a 21.6. ábrán láthatunk példát.

Nyilván több értelme lenne csak a törlés műveletre rákérdezni, és csak akkor, ha a mappa tartalmaz bizonyos fájlokat. Ám most, hogy már tudjuk, hogyan kell shellbe építeni megerősítést kezelő szervert, már kedvünk szerint programozhatjuk a Shellt. Ha vissza akarunk vonni egy megerősítés-kezelőt, akkor elég a Windows Registry HKEY_CLASSES_ROOT nevű részében a Directory\shellex\CopyHookHandlers-ben levő kulcslistából törölni. Újabb copy hook kezelő telepítése csak bővíti a listát, és nem töröl semmit.



21.6. ábra *A PasHook példa hatása, ha mappát próbálunk áthelyezni*

Hogyan tovább?

Ebben a fejezetben eleddig az OLE és a COM alapjaival foglalkoztunk. Láttuk, hogyan kell objektumokat elhelyezni DLL-ben, hogy kell őket COM objektumokká alakítani, és hogy hogyan kell olyan szervert készíteni, amely kommunikál a Windows 95 shell-lel.

A következő fejezetekben a Delphi 3 által közvetlenül támogatott szerverekkel foglalkozunk. Ilyenek az Automation szerverek, az Active Documentek (OLE dokumentumok), az ActiveX és az ActiveForm. Ezeket terjedelmi okokból nem ismertetjük részletekbe menően. Az érdekesség kedvéért bemutatom, hogyan kell OLE Automation szervereket és ActiveX kontrolokat írni Delphi 3-ban anélkül, hogy megismernénk azok működési mechanizmusát.

22. FEJEZET

OLE Automation és OLE Document technikák

E fejezet tartalma:

- Az OLE Automation
- OLE Automation szerver írása
- Az OLE dokumentum
- Többszörös OLE tárolók

Az OLE Automation és OLE Document technikák

Az ezt megelőző fejezetekben tisztáztuk az OLE és a COM alapjait. Ebben a fejezetben e technológia gyakorlati alkalmazásait kezdjük el vizsgálni. Delphiben nem kell feltétlenül ismerni a COM részleteit ahhoz, hogy alkalmazásokat írhassunk, mivel a környezet, és a rendelkezésre álló eszközök megóvnak bennünket a bonyodalmak nagy részétől. Ez különösen igaz arra a két témakörre, amit ebben a fejezetben tárgyalunk.

Első témánk az OLE Automation. Ez a név olyan szerver programokat takar, amelyek OLE interfészt szolgáltatnak ki kliensprogramoknak (OLE Automation irányítóknak). A 21. fejezetben a szerver kizárólag DLL lehetett, most azonban sima futtatható fájl lesz. Az OLE Automation támogatásában a Delphi 2 óta komoly fejlesztést vittek véghez. E fejezet másik témája az OLE dokumentum, illetve a Delphi OleContainer komponenseinek használata. Amint azt néhány példán keresztül látni is fogjuk, ez a komponens nem sokban különbözik a Delphi 2-béli társától.

Az OLE Automation

A Windows-ban a programok nem élnek önálló életet. A felhasználók gyakori igénye, hogy valahogy kommunikálni tudjanak egymással az alkalmazások. Amint majd a 30. fejezetben látni fogjuk a vágólap és a DDE teszi lehetővé a ezek kommunikációját. Egyre több program tartalmaz továbbá OLE Automation interfészt, ezáltal lehetővé téve, hogy más programok vezéreljék őt. Ennek az az előnye, hogy ezek az interfészek teljesen nyelvfüggetlenek, tehát használhatunk Delphit, C++-t, Visual Basic-et, vagy akár makrónyelvet is. Az OLE Automation szerverek vezérlésére független attól, hogy milyen nyelven írták.

Új!

A Delphi 2-ben az OLE Automation támogatása az OleAuto egység `TOleAuto` osztályán keresztül érhető el. Ez az egység megvan a Delphi 3-ban is a kompatibilitás végett, de az újabb, és hatékonyabb OLE Automation támogatás a ComObj egység `TOleAuto` osztályában található. Ebben a fejezetben főleg az új technológiát ismertetjük. A két említett egység többnyire kompatibilis, így elég könnyen át lehet írni a Delphi 2-béli példákat Delphi 3-ba.

Természetesen mi most arra fogunk koncentrálni, hogy hogyan kell Delphiben OLE Automation szervert használni, illetve saját szervert írni. Írunk egy OLE Automation irányítót is az OLE Automation szerverünk vezérlésére, elsősorban azért, mert nem tudom, hogy az olvasó milyen OLE Automation szervereket telepített a számítógépén. Az alapvető technikákat is bemutatom majd, egy egyszerű, Microsoft Wordhöz készült irányítón keresztül. A Microsoft Word egy eléggé elterjedt alkalmazás, így vélhetően mindenkinek megvan.

A Delphiben olyan egyszerű az OLE Automation használni, hogy használhatjuk kizárólagos eszközként is a programjaink kommunikációjának megvalósításához, még akkor is, ha más megoldások is kínálkoznának olyan esetekben, amikor mi írjuk mind a szerver, mind pedig a kliens programot.

Adatok küldése a Wordnek

Mielőtt nekiállnánk OLE Automation szerver programot írni, először hagy mutassak egy egyszerű Delphiben írt OLE irányítót. Ez a program kivesz néhány adatot egy adatbázisból, és készít belőle egy Microsoft Word dokumentumot úgy, hogy az OLE Automation interfészen keresztül vezérli a Wordöt.

A megvalósítás alapja, hogy deklarálunk egy variant típusú változót, és hozzárendeljük az OLE objektumot:

```
procedure TNavigator.BtnMasterWordClick(
    Sender: TObject);
var
  VarW: Variant;
begin
  VarW := CreateOleObject ('Word.Basic');
  VarW.FileNew;
  VarW.AppShow;
  VarW.Insert ('Mastering Delphi');
  VarW.FileSave;
end;
```

Figyelem!

A Microsoft Word számos különböző nyelven megjelent, továbbá a Microsoft némelyik OLE Automation interfészt is lefordította különböző nyelvekre, így elég nehéz általános érvényű kódot írni. A most következő példában a Word angol nyelvű verziójáról lesz szó, tehát előfordulhat, hogy a példa működik megfelelően a Word egyes változatain.

Ha még nem fut a szerver, akkor a CreateOleObjects hívás elindítja és visszaad egy referenciát, ami a kért objektumra mutat. Most már a WordBasic OLE szerver bármelyik tulajdonságához hozzáférünk, illetve bármelyik metódusát meghívhatjuk. Tehát a WordBasic Súgó fájlokban leírt összes makróparancsot használhatjuk. A fenti utasítások egyszerűen létrehoznak egy fájlt (ezt a részt ki is hagyhatjuk, ha a szöveget létező fájlba akarjuk behelyezni), a szervert előtérbe helyezik, beszúrnak egy sort, majd pedig elmentik a fájlt.

Ez utóbbi művelet megnyitja a Word Save as párbeszédablakát. Ha az AppShow hívást kihagytuk volna, akkor a párbeszédablak nem jelent volna meg, viszont az alkalmazás neve, és ikonja villogni kezdett volna, a párbeszédablak pedig csak azután jelent volna meg, amikor a felhasználó megnyitotta az alkalmazást. Általában nem ajánlatos a párbeszédablakokat láthatóvá tenni, mert ha a szerver nem aktív, akkor a háttérben maradnak. A fenti kódot azonban módosíthatjuk úgy, hogy a FileSave helyett a FileSaveAs-t hívjuk meg. A FileSaveAs hívás ugyanis közvetlenül elmenti a fájlt a paraméterként megadott névvel. Ha nem adunk meg paramétert, akkor a fájlt az alapértelmezés szerinti néven az alapértelmezés szerinti könyvtárba menti el.

Vegyük észre, hogy a metódus végén a variant változó megszűnik. Amint azt az előző fejezetből tudjuk, emiatt megszűnik az objektumhoz tartozó referencia. Ha a Word már fut, akkor ennek nincs túl sok hatása, ha viszont a Wordöt a CreateOleObject hívással nyitottuk meg, akkor mihelyst eltűnik a variant, a szerver azonnal leáll. Ez nem a normális bezárási művelet, ugyanis ilyenkor a szerver *terminál*. Ennek az egyik fontos tulajdonsága, hogy a dokumentumban végzett változásokat nem menti a program.

Adat küldése a Microsoft Wordnek egy adatbázisból

A fenti kód tulajdonképpen egy bonyolultabb példának a része. A példa neve DbToWord (ez a Database to Word rövidítése). Tulajdonképpen a 16. fejezetbeli Navigl példának egy újabb változata, ami egy olyan formot készített, amely egy rekordot mutat a COUNTRY.DB táblázatból. A kulcsfontosságú „Mastering Delphi –> Word" gombon kívül (amit az előbbi kód implementál) e program formja tartalmaz három új gombot. Az egyikkel a Wordből tudunk új példányt indítani, a másikkal a táblázatról jelentést küldeni neki, a harmadikkal pedig el tudjuk menteni a fájlt, mégpedig úgy, hogy előtte rákérdez az irányító alkalmazás (és nem a Word), hogy melyik mappába kívánjuk menteni dokumentumunkat.

A variant a formban külön mezőként van definiálva, így csak akkor nem látszik, és csak akkor áll le a szerver, amikor a irányító lefut. Az első gomb, az „Open Word" létrehozza a szerverobjektumot:

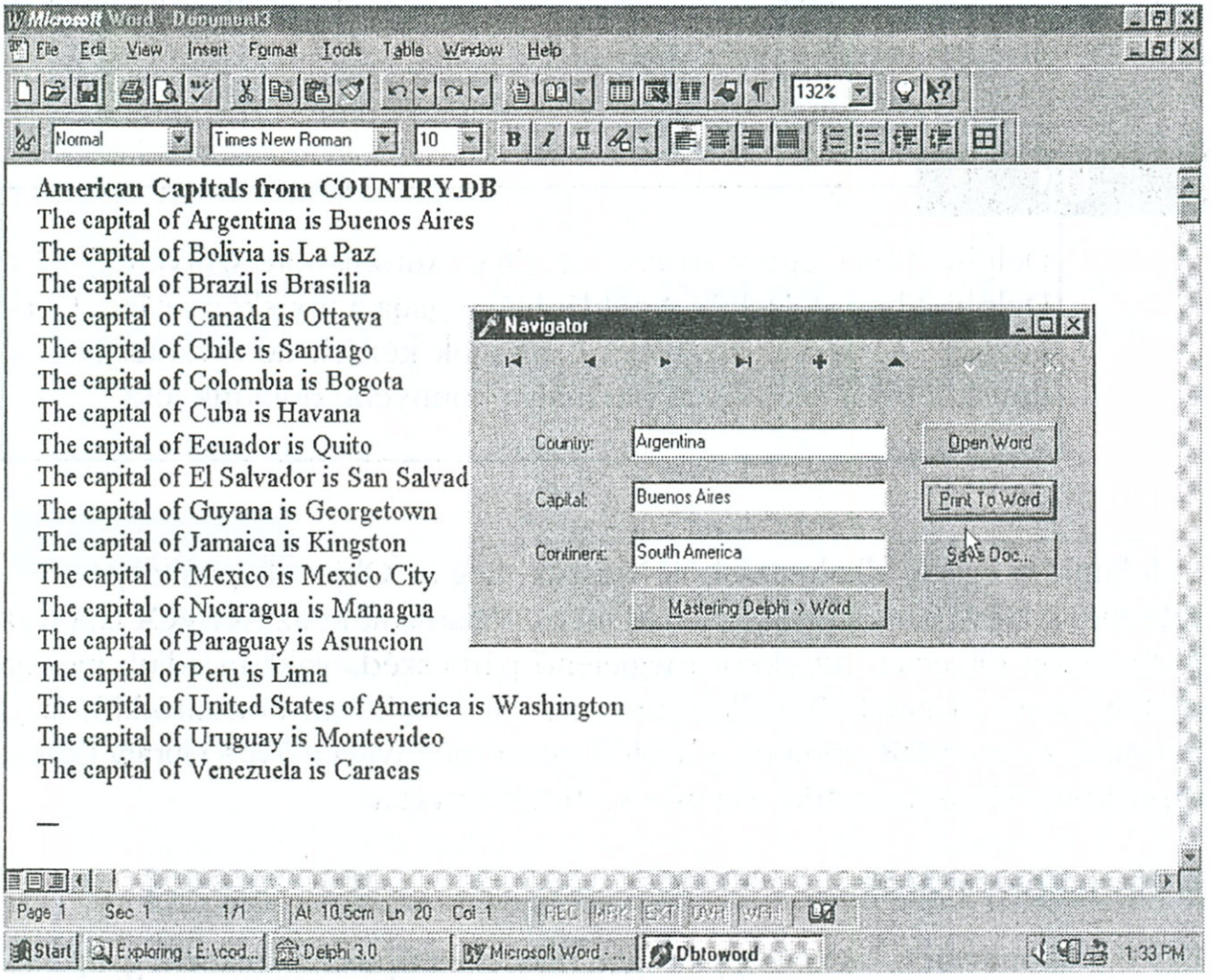
```
procedure TNavigator.BtnOpenClick(Sender: TObject);
begin
  // create the Automation object
  try
    VarWord := CreateOleObject ('Word.Basic');
    VarWord.AppShow;
    VarWord.FileNew;
    BtnPrint.Enabled := True;
    BtnSave.Enabled := True;
    // keep the program in front of Word
    Application.BringToFront;
  except
    ShowMessage('Microsoft Word not found');
  end;
end;
```

Ha a műveletet sikerült végrehajtani, akkor engedélyezi a másik két gombot, és visszahozza a programot a Word elé, hogy világossá váljon a felhasználó számára, hogy voltaképpen miről is van itt szó. Ha nem így történne, akkor a felhasználónak manuálisan kellene aktiválnia a programot, hogy a két gomb valamelyikét lenyomhassa. Ennek megfelelően a második gomb kódja a következő („Print to Word"):

```
procedure TNavigator.BtnPrintClick(Sender: TObject);
var
  Bookmark: TBookmark;
begin
  // disable the UI
  Table1.DisableControls;
  try
    // store the current position
    Bookmark := Table1.GetBookmark;
    try
      // insert the title (in bold font)
      VarWord.Bold;
      VarWord.Insert ('American Capitals from ' +
        Table1.TableName + #13);
      // scan the database table
      Table1.First;
      while not Table1.EOF do
      begin
        // send the two fields
        VarWord.Insert ('The capital of ' +
          Table1.FieldByName('Name').AsString+' is '+
          Table1.FieldByName('Capital').AsString+#13);
        Table1.Next;
      end;
```

```
        finally
          // go back to the bookmark and destroy it
          Table1.GotoBookmark (Bookmark);
          Table1.FreeBookmark (Bookmark);
        end;
      finally
        // re-enable the controls
        Table1.EnableControls;
      end;
    end;
```

Ez az eljárás beszúr egy kis vastag betűs címsort a dokumentumba, ami az adatbázis táblázatának nevét tartalmazza, majd az adattábla minden rekordját beszúrja a dokumentumba szövegként. Figyeljük meg a soremelés (#13) karakter használatát. Az eredmény a 22.1. ábrán látható.



22.1. ábra *A DbToWord nevű OLE Automation irányító formja és a Word dokumentum, amit a háttérben készített a program*

Az utolsó gomb (a „Save Doc") szabványos Delphi párbeszédablakot használ az útvonal és a fájlnév bekérésére:

```
procedure TNavigator.BtnSaveClick(Sender: TObject);
begin
  if SaveDialog1.Execute then
    VarWord.FileSaveAs (SaveDialog1.FileName);
end;
```

Delphi párbeszédablak használatával nagyobb rálátásunk van a felhasználó műveleteire. Ha viszont a felhasználó megnyomja a Cancel gombot a „Mastering Delphi" gomb által megjelenített Word File Save párbeszédablakon, a szerver hibaüzenetet küld.

OLE Automation szerver írása

A Delphi 3-ban nem csak OLE Automation irányítót könnyű írni, hanem OLE Automation szervert is. OLE Automation objektum létrehozásához használhatjuk a Delphi Automation Object Varázslót.

Új!

Delphi 3-ban egész mások az OLE Automation szerverek, mint Delphi 2-ben. A Delphi 3 például támogatja a típuskönyvtárak létrehozását. A szervert továbbra is írhatjuk kézzel, de a varázslót és a típuskönyvtár-szerkesztő használva könnyebb dolgunk lesz.

Induljunk ki egy új alkalmazásból. Nyissuk meg az Object Repository-t (ezt a **File** menü **New** parancsával tehetjük meg). Válasszuk ki az **ActiveX** lapon az Automation Objectet. Az ekkor megjelenő párbeszédablakban adjuk meg az osztály nevét (a kezdő T nélkül, mert az automatikusan hozzáíródik), majd nyomjuk meg az **OK** gombot. A Delphi most megnyitja a Type Library Editor-t (típuskönyvtár-szerkesztőt). De mi az a típuskönyvtár?

A típuskönyvtárak

Egy típuskönyvtár tulajdonképpen nem más, mint típusinformációk gyűjteménye. Egy ilyen gyűjtemény általában a szerver által láthatóvá tett elemek (objektumok, interfészek) információit tartalmazza. A fő különbség a típuskönyvtár és ezen elemek más jellegű leírásai között az, hogy a típuskönyvtár nyelvfüggetlen. A típuselemeket az OLE a szabványos programozási elemek részhalmazaként definiálja, így azok minden fejlesztőeszközzel használhatóak. De miért kell nekünk ez az információ?

Az imént készített OLE Automation irányító nem rendelkezett semmilyen típusinformációval az általa használt szerverről. Ez azt jelenti, hogy a háttérben minden függvényhívás az IDispatch.Invoke-on keresztül jutott el a szerverhez úgy, hogy a függvény nevét string paraméterként adta át a kliens, és remélte, hogy a név megegyezik valamilyen létező függvénynévvel. A fordító sehogy sem tudta ellenőrizni, hogy a metódus létezik e. A típusellenőrzéseket futásidőben végrehajtani kockázatos, hiszen ha akár egyetlen apró elírás is van az egyik függvénynévben, akkor semmi figyelmeztetést nem kapunk a hibáról egészen addig, amíg a program futás közben el nem jut az adott sorig.

Az OLE IDispatch interfész támogatja ezt a fajta megközelítést is, ám a szerver az interfészeire és objektumaira vonatkozó információt típuskönyvtáron keresztül is exportálhatja. A típuskönyvtárat aztán a megfelelő eszközzel átalakíthatjuk az általunk használni kívánt nyelv szintaxisának megfelelő definíciókká, így a fordítóprogram ellenőrizheti, hogy a kód megfelelő e.

Ha a fordítóprogram elvégezte az ellenőrzéseket, akkor két különböző technikával is el tudja juttatni a kérést a szerverhez. Használhat sima VTable-t (azaz egy bejegyzést az interfész típusdeklarációjában), de használhat `dispinterface`-t is. Interfész típusdeklarációt már kipróbáltuk az előző fejezetben, így az bizonyára ismerős, a `dispinterface` viszont valami egészen új dolog. Ez tulajdonképpen egy módszer arra, hogy egy interfész bejegyzéseit számokká képezzük le. Ezek után pedig a szerverre érkező hívásokat lehet szám szerint azonosítani. Tekinthetjük ezt a technikát átmenetnek is a név szerinti hivatkozás, és a közvetlen VTable hivatkozás között.

Megjegyzés

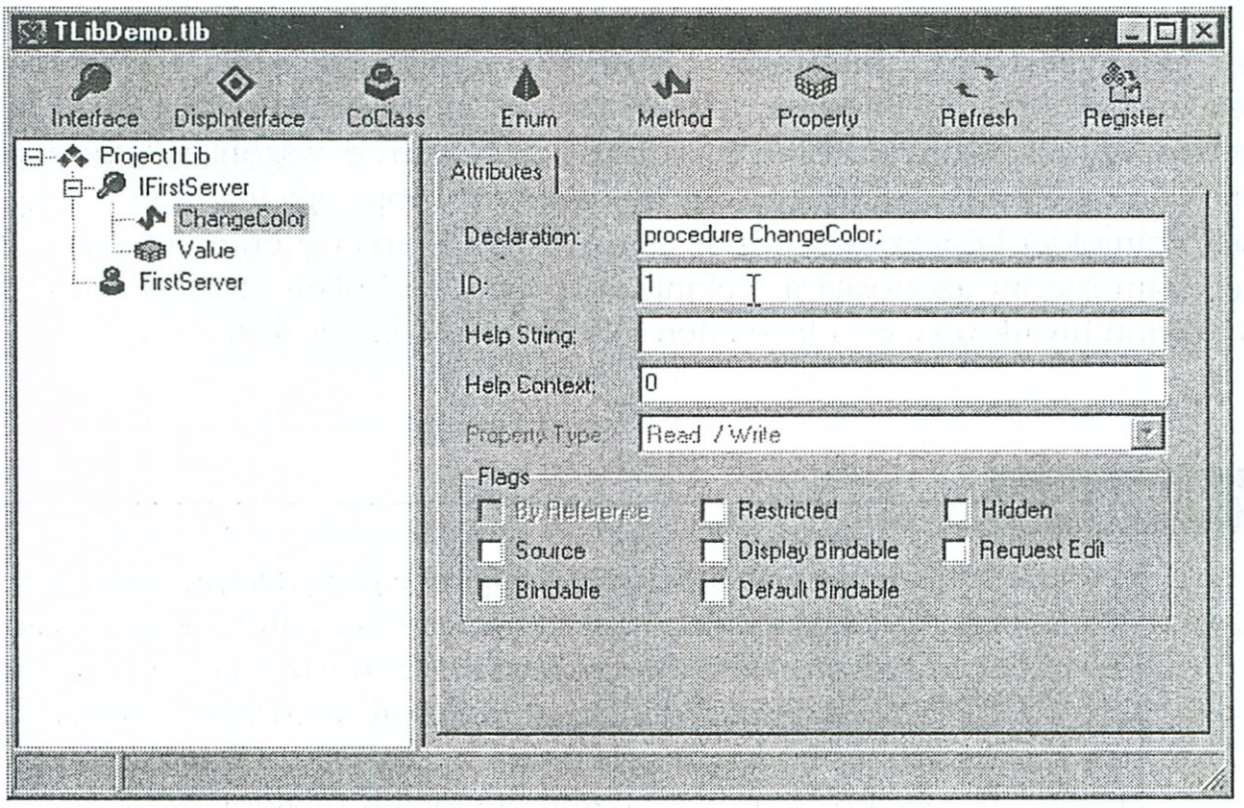
A `dispinterface` szó tulajdonképpen egy új Delphi kulcsszó. Nem fogom részletesen ismertetni, de mutatok egy olyan `dispinterface`-t, amit a típuskönyvtár-szerkesztő generált. Ezen kívül a Delphi 3-ban más ide kapcsolódó kulcsszavakat is bevezettek: a `dispid` az egyes elemekhez rendelt számot jelöli, a `readonly` és a `writeonly` pedig opcionális tulajdonságok.

Ezt a lehetőséget, hogy egy szerverrel két különböző (egy inkább dinamikus, és egy inkább statikus) módon lehet kapcsolatba lépni, a duális interfészek technológiájának nevezzük. Ez a két mód a következő: használhatja a késői kötést (late binding), és a `dispinterface` által kínált mechanizmust, vagy használhat korai kötést (early binding), és a VTable-ön alapuló mechanizmust.

Viszont ne feledkezzünk meg arról sem, hogy a különböző technikák végrehajtási sebessége is más és más. A függvények név szerinti kikeresése (és a típusellenőrzés futásidejű végrehajtása) a létező leglassabb módszer. A `dispinterface` használata sokkal gyorsabb, a közvetlen VTable hivatkozás pedig a leggyorsabb.

A típuskönyvtár-szerkesztő

Most menjünk vissza a példánkhoz, és használjuk a típuskönyvtár-szerkesztőt (Type Library Editor). A 22.2. ábrán látható a program ablaka néhány bemenő adat befogadása után. A típuskönyvtár-szerkesztő lehetővé teszi, hogy metódusokat és tulajdonságokat adjunk hozzá a most készített OLE Automation szerverobjektumhoz. Ha ez is kész van, akkor a típuskönyvtár-szerkesztő létre tudja hozni a típuskönyvtár fájlt és a hozzá tartozó Object Pascal forráskódot is.



22.2. ábra *A típuskönyvtár-szerkesztő*

Vegyünk egy egyszerű példát. A szerverhez egy tulajdonságot és egy metódust akarunk hozzávenni. A szerkesztőben vegyük hozzá ezt a két elemet az interfészhez. Az interfész neve legyen IFirstServer. Egyszerűen válasszuk ki, majd nyomjuk meg a Method gombot. Most már csak el kell nevezni, például így:

```
procedure ChangeColor;
```

Most nyomjuk meg a Property gombot, és adjunk neki nevet, valamint adattípust:

```
property Value: Integer;
```

Megjegyzés

Ez az OLE Automation interfész bármely nyelven írt program számára elérhető kell hogy legyen, ezért aztán a tulajdonságokban, paraméterekben és visszatérési értékekben az OLE által támogatott adattípusokra kell szorítkoznunk. Az OLE a következő alapvető típusokat támogatja: Integer, SmallInt, Byte, Single, Double, WideString, Variant és WordBool, viszont nem támogatja a Boolean típust.

Most már elmenthetjük a változtatásokat, és bezárhatjuk a típuskönyvtárat. Ez a művelet három új elemet ad hozzá a projekthez: a típuskönyvtárat, a hozzá tartozó Pascal definíciót, valamint a szerverobjektum deklarációját.

A típuskönyvtár kapcsolatban áll azzal az erőforrás-használatra vonatkozó kódrészlettel, ami most került bele a projektfájl forráskódjába:

```
{$R *.TLB}
```

A típuskönyvtár-szerkesztőt mindig újra és újra megnyithatjuk, ha kiválasztjuk a megfelelő TLB fájlt a Delphi File Open párbeszédablakában.

A típuskönyvtár ezután átalakul interfész definícióvá, majd bekerül az új Pascal egységbe. Ez az egység elég hosszú, ezért csak a fontosabb elemei kerültek bele ebbe a könyvbe. Az itt következő rész az új interfész deklarációja:

```
type
  IFirstServer = interface(IDispatch)
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; safecall;
    function Get_Value: Integer; safecall;
    procedure Set_Value(Value: Integer); safecall;
    property Value: Integer read Get_Value
        write Set_Value;
  end;
```

Ez az osztály egyszerűen csak listázza a típuskönyvtár-szerkesztővel vizuálisan felépített elemeket. A két plusz metódus a tulajdonság definíciója miatt kell, az előző fejezetben elmondottak alapján.

Megjegyzés

Ennek az interfésznek a metódusai a `safecall` hívási konvenciót használják. Ez annyit jelent, hogy levédik az egyes metódusokat egy-egy `try-except` kivételkezelő blokkal, és ahogy az előző fejezetben láttuk, beállítanak alapértelmezett visszatérési értéket is.

Következzék a `dispinterface`, ami számokat rendel az IFirstServer interfész elemeihez:

```
type
  DFirstServer = dispinterface
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; dispid 1;
    property Value: Integer dispid 2;
  end;
```

E fájl minden deklarációja (van még néhány, amit kihagytam) tekinthető belső, rejtett implementációs támogatásnak is. Nem kell teljesen megérteni, és nem szabad közvetlenül szerkeszteni őket.

Végül pedig a Delphi generál egy fájlt a tényleges objektum deklarációjával. Ez az egység az alkalmazás részévé válik, és ez lesz az az egység, amivel dolgoznunk kell ahhoz, hogy befejezzük a programot. Ez az egység deklarálja a szerverobjektumnak azt az osztályát, ami tartalmazza a most definiált interfészt:

```
type
  TFirstServer = class(TAutoObject, IFirstServer)
  protected
    function Get_Value: Integer; safecall;
    procedure ChangeColor; safecall;
    procedure Set_Value(Value: Integer); safecall;
  end;
```

A Delphi megadja a metódusok kódjának vázlatát is, így csak ki kell tölteni a köztes sorokat. Íme a kész kód a TLibDemo példabeli szerverobjektum metódusainak:

```
function TFirstServer.Get_Value: Integer;
begin
  Result := Form1.Value;
end;

procedure TFirstServer.ChangeColor;
begin
  Form1.ChangeColor;
end;
```

```
procedure TFirstServer.Set_Value(Value: Integer);
begin
  Form1.Value := Value;
end;
```

Ebben az esetben a három metódus egyszerűen csak hivatkozik egy tulajdonságra, és két metódusra a formból. Általában szerintem nem helyes a szerverobjektum osztályaiba a felhasználói felületre vonatkozó kódot írni. Jobban tesszük, ha inkább hivatkozunk egy felhasználói interfészbéli elemre, mondjuk egy form osztályra, és azzal végeztetjük el a feladatot.

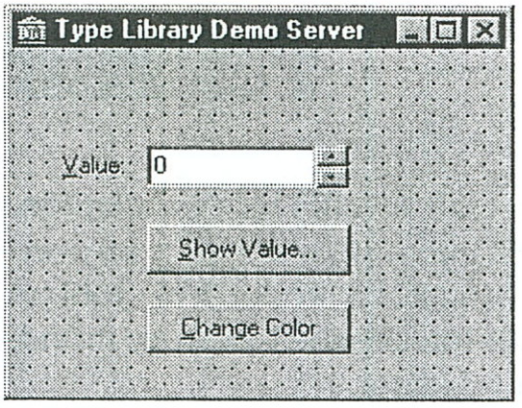
Hozzávettem egy új tulajdonságot a formhoz, mert meg akartam változtatni a Value tulajdonságot, és ezzel előcsalni egy mellékhatást is (ami az, hogy az új érték megjelenik az szerkesztőmezőben). A másik megközelítés, ami az lenne, hogy magába a szerver objektumba írjuk bele a mellékhatást produkáló kódot, szerintem túlontúl bonyolult. A szerverobjektum ebben a példában egyszerűen csak láthatóvá teszi az alkalmazás néhány tulajdonságát és metódusát. Itt olvasható a TForm1 osztály deklarációjának általam írott része:

```
type
  TForm1 = class(TForm)
    ...
  private
    CurrentValue: Integer;
  protected
    procedure SetValue (NewValue: Integer);
  public
    property Value: Integer read CurrentValue
        write SetValue;
    procedure ChangeColor;
  end;
```

Ezen metódusok megvalósítása elég egyértelmű, és könnyen kitalálható, hogy miként fest a kódjuk. A fontos rész viszont a SetValue metódus, aminek lehet mellékhatása is:

```
procedure TForm1.SetValue (NewValue: Integer);
begin
  if NewValue <> CurrentValue then
  begin
    CurrentValue := Value;
    UpDown1.Position := CurrentValue;
  end;
end;
```

Ennek a példának a formja tartalmaz egy szerkesztőmezőt, amelyhez tartozik egy léptetőgomb is, meg néhány gomb, amelyekkel megnézhetjük az aktuális értékeket, és megváltoztathatjuk a színt. A 22.3. ábrán látható a tervezés alatt álló form.



22.3. ábra *A TLibDemo formja kirajzolás közben*

Az Automation szerver regisztrálása

A szerverobjektumot tartalmazó egységben van még egy kódrészlet, amit a Delphi adott hozzá az inicializációs részhez:

```
initialization
  TAutoObjectFactory.Create(ComServer, TFirstServer,
    Class_FirstServer, ciMultiInstance);
end.
```

Ez nem sokban különbözik az előző fejezetben látott osztálygyáraktól. Annyiban tér csak el, hogy tartalmaz egy hívást az Application objektum Initialize metódusára. Ezt a Delphi alapértelmezésben hozzáadja minden program projekt forráskódjához. A fenti inicializációs kód teszi a szerver regisztrációját egyértelművé.

A szerverre vonatkozó információt úgy vehetjük bele a Windows Registry-be, ha a `/regserver` paraméterrel lefuttatjuk az alkalmazást a célgépen (azon a gépen, ahol telepíteni akarjuk az OLE Automation szervert). Ezt úgy is megtehetjük, ha a Start menüből kiválasztjuk a Futtatás menüpontot, és használjuk az Intézőt illetve a Fájlkezelőt, vagy ha egyszerűen úgy futtatjuk a programot Delphiben, hogy előzőleg megadjuk a parancssor paramétert (a **Run** menü **Parameters** parancsával). Egy másik parancssor paraméter (`/unregserver`) segítségével pedig eltávolíthatjuk a szervert a Registry-ből.

Írjunk klienst a szerverünkhöz!

Most, hogy építettünk egy szervert, már készíthetünk kliens programot is, amivel leteszteljük. Persze használhatnánk ugyanazt a megközelítést és kódot, mint amit a fejezet eredeti példaprogramjaiban alkalmazhatunk. Viszont mivel most rendelkezésünkre áll a szerverhez tartozó típuskönyvtár, ezért ezúttal más megközelítést is választhatunk.

Készítsünk egy új alkalmazást – én ezt TLibCli-nek neveztem – majd nyissuk meg a szerver típuskönyvtárát, miután (opcionálisan) bemásoltuk a projekt könyvtárába. Elég elmenteni a típuskönyvtár fájlt a Delphi **File** menüjének **Save** parancsával, és az interfészdeklaráció új verziója automatikusan létrejön. Persze a mi esetünkben elég lett volna átmásolni a Pascal deklarációt a szerver forráskódjából, de most igyekszem minél általánosabban alkalmazható megoldásokat bemutatni, amelyeket akkor is használhatóak, ha más valaki írta a szervert.

Ne rakjuk viszont bele a típuskönyvtárat az alkalmazásba, mert most OLE Automation irányítót írunk, és nem szervert. Az irányító Delphi projektje tartalmazhat típuskönyvtárat, de az a szerver típuskönyvtára lesz.

A főform kódjában egyszerűen hivatkozhatunk a típuskönyvtár-szerkesztő által létrehozott Pascal fájlra:

```
uses
  Project1Lib;
```

Ez az egység több elemet is definiál. Ezek egyike a creation osztály. Ez egy olyan speciális osztály, amely két osztályfüggvényt (vagy statikus metódust, lásd 5. fejezet) tartalmaz:

```
type
  CoFirstServer = class
    class function Create: IFirstServer;
    class function CreateRemote(
      const MachineName: string): IFirstServer;
  end;
```

Az első függvényt – a Create-et – arra lehet használni, hogy létrehozzunk a saját gépünkön egy szerverobjektumot (és lehetőség szerint elindítsuk a szerveralkalmazást). A második függvénnyel – a CreateRemote-tal – egy másik számítógépen hozhatunk létre szervert, már ha egyáltalán az operációs rendszerünk támogatja a távoli OLE Automationt, vagy más szóval a DCOM-ot (Distributed COM).

A most következő példákban az első függvényt használom. Ha létrehozzuk a szerverobjektumot, akkor visszatérési értékként egy IFirstServer interfészt kapunk. Ezt felhasználhatjuk közvetlenül is, de el is tárolhatjuk egy variant típusú változóban. Íme egy példa az elsőként említett módszerre:

```
var
  MyServer: Variant;
begin
  MyServer := CoFirstServer.Create;
  MyServer.ChangeColor;
```

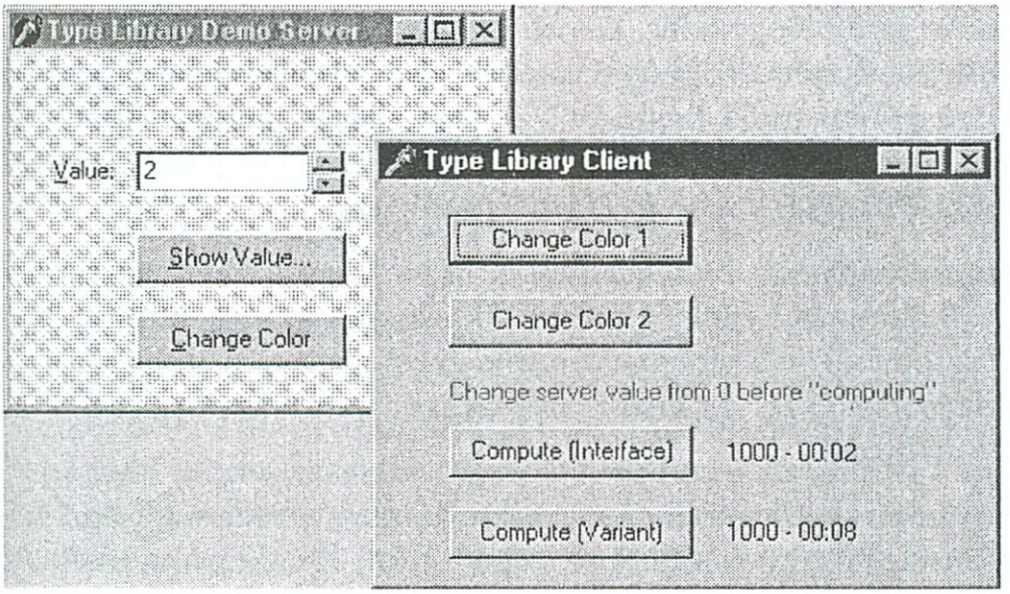
Ez a variantokra épülő kód nem sokban különbözik a fejezet elején lévő irányítók kódjától. Alább látható a másik lehetséges kód, aminek tökéletesen ugyanaz a hatása:

```
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
```

Amint azt már a típuskönyvtárakat bemutató részben említettem, a két módszer közötti egyik legszembetűnőbb különbség a végrehajtási sebesség. Elég bonyolult lenne nyomon követni a technika pontos működését, hiszen itt sok tényező játszik szerepet. A TLibCli példa tartalmaz egy egyszerű tesztet, csak hogy legyen némi fogalmunk a különbségről. Most a teszt kódja következik. Ez egy olyan ciklus, ami a szerver Value tulajdonságát olvassa be kétszer. Ha ezt a két értéket elosztjuk egymással, akkor az eredmény azonosan 1 lesz, így aztán a K végső értéke a ciklusmag lefutásainak számát tartalmazza. Ez az érték jelenik meg aztán egy címkében az eltelt idővel együtt:

```
procedure TForm1.BtnIntfClick(Sender: TObject);
var
  I, K: Integer;
  Init: TDateTime;
begin
  Screen.Cursor := crHourglass;
  try
    Init := Now;
    K := 0;

    for I := 1 to 1000 do
      K := K + IMyServer.Value div IMyServer.Value;

    Label1.Caption := IntToStr (K) + ' - ' +
      FormatDateTime ('nn:ss', Now - Init);
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

A 22.4. ábra szemlélteti, hogy mi lesz a kimenet, ha meghívjuk ezt a metódust, illetve ha a program variantra épülő változatát használjuk. Természetesen az időzítés a számítógép sebességétől is függ, és az eredményt úgy is megváltoztathatjuk, ha növeljük vagy csökkentjük a ciklusszámláló maximális értékét. Mielőtt megnyomnánk a két Compute gomb valamelyikét előbb menjük át a szerverre, és állítsuk a Value tulajdonságot valami nullától különböző értékre.



22.4. ábra *A TLibCli OLE Automation irányító kétféleképpen érheti el a szervert. Figyeljük meg a háttérben lévő szerverablakot!*

Az OLE dokumentum

Mint már az előző fejezetben említettem, az OLE eredeti formáját újabban OLE dokumentumnak hívják. Ezek az eljárások hajtódnak végre, például amikor külső szerverről hozunk be egy objektumot Wordbe vagy Excelbe. Az OLE dokumentumok két módon illeszkednek be, beágyazással vagy csatolással:

- A beágyazás az OLE dokumentumoknál megfelel a Vágólap Másol–Beilleszt műveletsorának. A fő különbség az, hogy amikor OLE objektumot másolunk egy szerveralkalmazásból, és beillesztjük a kliensalkalmazásba, akkor az adaton kívül a szerverre vonatkozó információkat is átviszünk (a GUID-jét). Ezáltal a szerveralkalmazás a kliensből is aktiválható lesz. Erre akkor lehet szükség, ha esetleg szerkesztési műveleteket kell végezni az adatokon.
- A csatolás ezzel szemben mindössze egy referenciát másol át, ami az adatokra, és a szerverre vonatkozó információkra mutat. A csatolást általában úgy aktiváljuk, hogy a vágólapot használva végrehajtjuk a Beillesztés–Csatolás műveletet. Amikor a tároló alkalmazásban szerkesztgetjük az adatot, akkor tulajdonképpen az eredeti adatot manipuláljuk, ami egyébként egy külön fájlban van.

Ezért van, hogy ilyenkor a szerver csak várakozó (stand-alone) ablakban aktiválódik, és az egész eredeti fájllal dolgozik, nem pedig csak az átmásolt adatokkal. Ezzel szemben, amikor beágyazással illesztünk be objektumot OLE 2.0 szerverről, akkor a tároló alkalmazás lehet, hogy támogatja a vizuális (vagy in-place) szerkesztést, ami annyit jelent, hogy az objektumot a tároló (beágyazó)

alkalmazás főablakán belül végezzük. Ez esetben a „gazdaprogram" menüi, eszköztárai átalakulnak a szerver igényei szerint, így a felhasználó a szerver programot teljes egészében tudja használni az aktuális programon belül. Ez lehetőséget biztosít egyetlen ablakon belül több különböző objektumtípussal (s ezáltal több különböző OLE szerverrel) való munkára.

A másik különbség a beágyazás és a csatolás között, hogy az beágyazott objektumot a tároló alkalmazás kezeli, a tároló alkalmazás az objektumot saját fájljába menti. A csatolással másolt objektumok ezzel szemben külön fájlban vannak, amelyet kizárólag a szerver kezel, még akkor is, ha a kapocs csak a fájl egy apró részére vonatkozik.

A tároló alkalmazásnak egyik esetben sem kell tudnia a szerver segítsége nélkül kezelni, illetve megjeleníteni az adott objektumot, illetve annak adatait. Emiatt aztán a szerver alkalmazásnak még akkor is rengeteg a tennivalója, ha éppen nem szerkesztjük az adatokat. A tároló alkalmazások gyakran másolatot készítenek az OLE objektum képéről, és az így kapott bitképet használják az adatok megjelenítésére. Ezáltal az objektummal végzett bizonyos műveletek gyorsabban hajthatók végre. E módszer nagy hátránya, hogy minden adat a gazdaprogram által készített fájlban kerül tárolásra, így az nagyon nagy lehet. Ha e problémához hozzávesszük, hogy az OLE aránylag lassú technológia, és fejlesztése munkaigényes, akkor világossá válik, hogy miért nem hódított akkora teret, mint amekkorát a Microsoft néhány éve jósolt neki.

Alapjában véve kétféle OLE dokumentum szerver létezik:

- A teljes szerverek önálló alkalmazások, saját fájlkezeléssel, támogatják az OLE minden formáját.
- A mini szerverek a DLL-ekben vannak. Ezek a DLL-ek önmagukban nem futtathatóak, nem készítenek saját fájlokat, a hívó alkalmazás által biztosított adatterületen dolgoznak. A mini szerverek továbbá kizárólag a beágyazást támogatják, a csatolást nem.

Másrészt az OLE dokumentumok tárolói különböző mértékben támogatják az OLE-t. Még ha nem vesszük figyelembe a verziók közti különbségeket, akkor is számos különböző módszer kínálkozik arra, hogy egy objektumot elhelyezzünk egy tároló alkalmazásban:

- Egy menüparancs segítségével új objektumot szúrhatunk be beágyazással (embedding).
- Beilleszthetünk a vágólapról is egy objektumot a Beillesztés paranccsal, ezáltal létrehozva egy új objektumot (embedding).
- A beillesztés-csatolás paranccsal is hozhatunk be objektumot a vágólapról. Ezzel csatolt objektumot hozunk létre (linking).

- Készíthetünk új objektumot a szerver által készített fájlokból. Ezt a csatolással és a beágyazással egyaránt megtehetjük.
- Ha mindkét alkalmazás támogatja a fogd és vidd (drag-and-drop) technológiát, akkor ezzel a módszerrel is vihetünk át objektumot a szerverből a tároló alkalmazásba. Objektum belsejében is hozhatunk létre objektumot, ha a szerver alkalmazás tud tárolóként működni.

Ha elhelyeztünk egy objektumot a tároló alkalmazásban, akkor műveleteket végezhetünk rajta a szerver elérhető műveleteinek segítségével. Általában az szerkesztés az alapértelmezett művelet (az alapértelmezett művelet itt az a művelet, ami akkor hajtódik végre, ha duplán kattintunk rá az objektumra egérrel). A többi objektumra a Lejátszás az alapértelmezett művelet. Általában úgy nézhetjük meg, hogy milyen műveleteket támogat az aktuális tároló, ha az egér jobbgombjával rákattintunk. Ugyanezt az információt találjuk meg sok programban a Szerkesztés menü Objektum pontjában, ami almenü formájában mutatja az adott objektumhoz tartozó lehetséges műveleteket.

Az OleContainer komponens használata

A Delphi nem támogatja az OLE dokumentum szerverek vizuális szerkesztését, mindig meg kell írni a megfelelő interfészeket tartalmazó szervert. Az OLE dokumentumok tárolójának támogatása viszont egyszerűen elérhető az OleContainer komponensen keresztül. E komponens használatával egy példa fokozatos felépítésével ismerkedünk meg. A példa tartalmazza a fentiekben ismertetett OLE tároló technikák nagy részét.

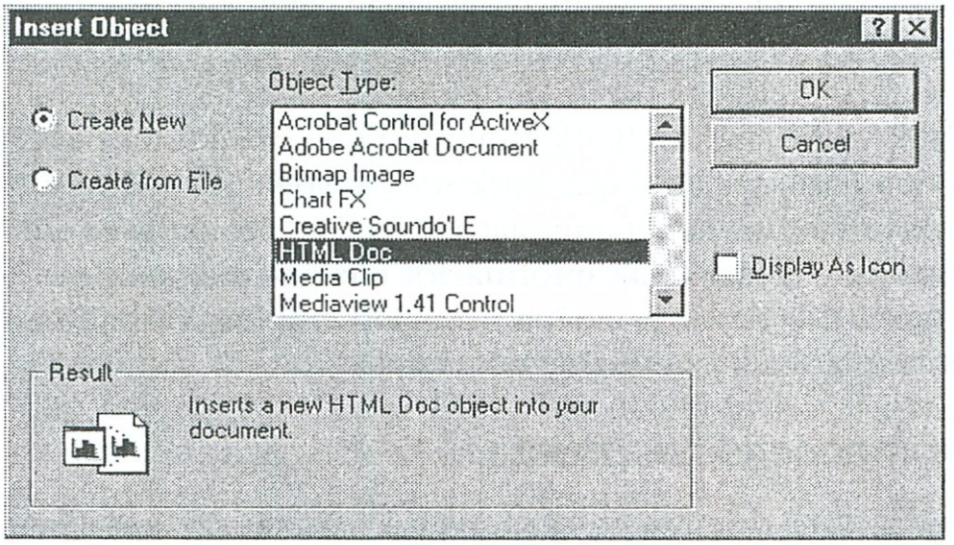
Megjegyzés

Az OLE tárolók támogatása a 32-bites Delphiben sokat változott a 16-bites Delphihez képest. A komponens neve ugyanaz, de az őt definiáló egység megváltozott. Az OLE segédfüggvényei megszűntek (helyettük viszont ott vannak a TOleContainer osztály új metódusai). Másrészt az új OLE objektumok létrehozása is egész másképp történik.

Egy egyszerű OLE tároló

Delphiben úgy készíthetünk a legegyszerűbben OLE tároló alkalmazást, ha elhelyezünk egy OleContainer komponenst egy formban. Ezek után válasszuk ki a komponenst, és kattintsunk rá a jobb gombbal, ezzel aktiválva a lokális menüjét. Ebben a menüben lesz egy Insert Object parancs. Ha kiválasztjuk ezt a

parancsot, akkor a Delphi megjeleníti a szabványos OLE Insert Object (objektum beszúrás) párbeszédablakot, ezen keresztül kiválaszthatjuk a regisztrált szerveralkalmazások valamelyikét. Ezt illusztrálja a 22.5. ábra. Hogy a párbeszédablakban megjelenő lista mely szervereket tartalmazza, az attól függ, hogy milyen OLE alkalmazások és OLE vezérlők érhetőek el a rendszerben, és ezek közül melyek szerepelnek a regisztrációs adatbázisban.



22.5. ábra *A szabványos OLE Insert Object párbeszédablak segítségével a rendszerünkben telepített szerverek bármelyikéről hozhatunk be OLE objektumot*

Ha az OLE objektumot betettük a tárolóba, akkor a tároló komponens helyi menüje néhány újabb menüponttal bővül. Az újabb menüpontok segítségével megváltoztathatjuk az OLE objektum tulajdonságait, beszúrhatunk újabb OLE objektumot, ezen kívül létező OLE objektumot másolhatunk, valamint törölhetünk.

A lista az objektum műveleteit is tartalmazza (pl. Szerkesztés/Edit, Megnyitás/Open, Lejátszás/Play). Ha belerakunk egy OLE objektumot egy tárolóba, akkor a hozzá tartozó szerver indul el, mikor szerkeszteni kívánjuk az új objektumot. Mihelyst bezárjuk a szerver alkalmazást, a Delphi frissíti a tárolóban lévő objektumot, majd megjeleníti a fejlesztés alatt álló Delphi alkalmazás formjában.

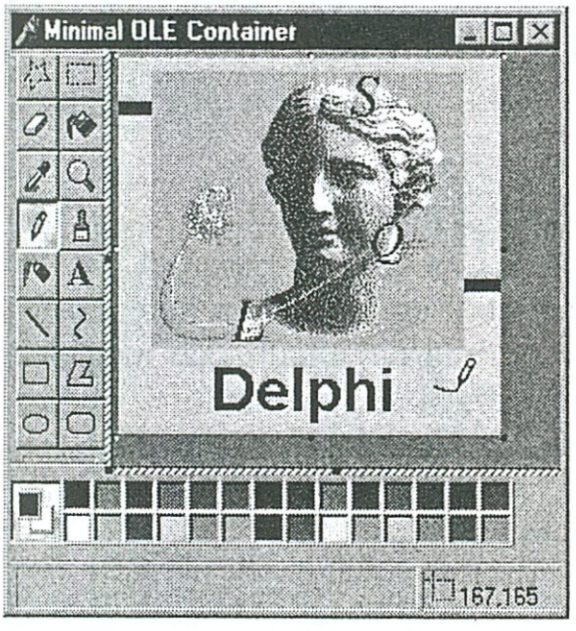
Ebben az OleContl programban semmi más jelentős változtatást nem vittem véghez azon kívül, hogy átraktam az OLE tárolót a kliens területre, és átállítottam a SizeMode tulajdonságát `smCenterre` (hogy az OLE objektum a tároló komponens közepén jelenjen meg). Ha vetünk egy pillantást ennek a formnak a szöveges jellemzésére (a form lokális menüjének View as Text parancsával), akkor feltűnik, hogy van egy Data tulajdonság is, ami az OLE objektum

tényleges adatát tartalmazza. A kliens program csak tárolja az objektum adatait, de nem tudja őket kezelni és megjeleníteni a megfelelő szerver nélkül. Más szóval az OLE objektum az beágyazással került be a tárolóba. A beágyazott OLE objektum a form belsejében tárolódik el, ami pedig a program futtatható állományába kerül. Ez nagymértékben megnövelheti a program méretét a szerver típusától és az adat mennyiségétől függően. A másik baj az, hogy ezt a programot csak olyan számítógépen lehet futtatni, amelyiken telepítve van a megfelelő szerver. Ha ugyanis ez hiányzik, akkor futásidejű hiba keletkezik, amikor az OLE tároló megpróbálja elindítani a szerveralkalmazást.

Megjegyzés

Ez az OleCont1 példaprogramra is igaz. Csak akkor futtathatjuk, ha a Microsoft Paint szerver telepítve van a gépünkön (ez a szerver a Windows alapértelmezett része, általános telepítés során ez a program is telepítésre kerül).

A 22.6. ábrán látható, hogy az OleContl példa futtatásakor elég egyszerűen duplán kattintani az OLE tároló komponensen, és a szerver ablaka megjelenik az alkalmazásunk belsejében.



22.6. ábra *Az OleCont1 példa futás közben. Itt látható működés közben a vizuális szerkesztés*

Az objektumszerkesztő akkor aktivizálódik, ha duplán kattintunk az egérrel a komponensre. Ez az AutoActivate tulajdonság értéke miatt van így. E tulajdonság értéke alapértelmezésben `aaDoubleClick`. A másik két lehetőség az `aaGetFocus` és az `aaManual`. Ha az érték `aaManual`, akkor kódból is elindíthatjuk a szervert az Active tulajdonság segítségével. Ha az érték `aaGetFocus`, akkor a szerver azonnal elindul, mihelyst ráirányítjuk az input fókuszt (azaz például rákattintunk az egérrel).

Megjegyzés

Nem használható mindig az Active tulajdonság a szerkesztés lezárásához. Ha a szervernek saját ablaka van, akkor azt a felhasználónak kell bezárnia.

Ha a beágyazott szerkesztésnél szeretnénk a szerver menüit is importálni és a felhasználónak felkínálni, akkor bővítsük ki az alkalmazást egy menüsorral. A most következő részben ennek megvalósítását mutatom be.

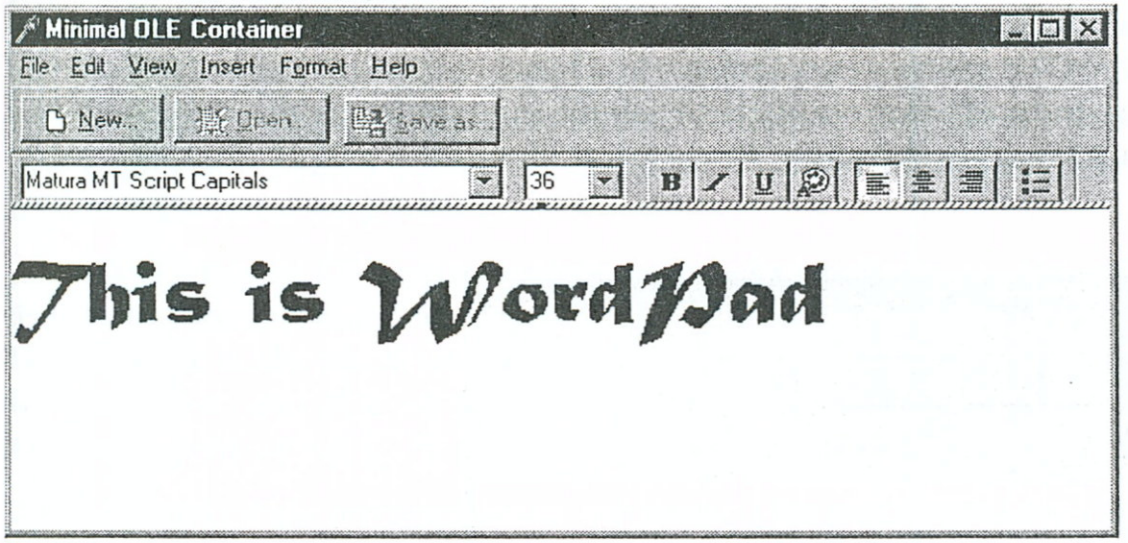
OLE tároló bővítése menüsorral

Készítettem egy új formot, hogy az eddigiekben megismert OLE tároló alkalmazást fejleszteni tudjuk. Az új form tartalmaz egy OLE tároló komponenst, egy menüt, és egy panelt, amin pedig egy kombinált lista van. Az új alkalmazás neve OleCont2. Ebben a változatban a tároló nem tartalmaz OLE dokumentumot a megjelenítés pillanatában. Az új OLE objektum akkor jön létre, amikor a File menüből kiválasztjuk a New parancsot.

Az OLE tárolók egyik legfontosabb tulajdonsága a menüátvitel. Ez szolgáltatja a menüket, és támogatja a helyi szerkesztést. Ha az OLE objektumot helyben aktiváljuk, akkor a szerveralkalmazás menüsorából egyes menük átkerülnek a tároló alkalmazás menüsorába. Ezzel egy időben a tároló alkalmazás megfelelő menüi eltűnnek. Az OLE menüátvitelt a Delphi szinte automatikusan kezeli. Mindössze annyit kell tennünk, hogy a GroupIndex tulajdonság segítségével beállítjuk a tároló menüpontjainak indexeit. A páratlan indexű menüpontok cserélődnek le az aktív OLE szerver megfelelő elemeire.

Még specifikusabban a File (0) és a Window (4) menük tartoznak a tároló alkalmazáshoz, az Edit (1), a View (3), és a Help (5) menük (illetve az ilyen indexű menük) pedig az OLE szervertől jönnek. A hatodik csoport – melynek neve Object, indexe pedig 2 – arra használható, hogy új menüt hozzunk létre

az Edit és a View csoportok között. Ez akkor is működik, ha az OLE objektum inaktív. A mi példánkban a GroupIndex tulajdonságok a következők: Fájl - 0, Edit - 1, Help - 5. A 22.7. ábrán láthatunk példát a menüátvitelre egy Paint OLE objektum esetében.



22.7. ábra *Az aktív OLE objektum szervere által felépített menüsor az OleCont2 példában (csak a File menü származik a tároló alkalmazásból)*

Csak az egyik menüpont kódját kell megírnunk: a New parancsnak, aminek a funkciója új OLE objektumok beszúrása a tárolóba. Ehhez felhasználhatjuk az InsertObjectDialog metódust a `TOleContainer` osztályból, ami megjelentíti a szabványos Insert OLE Object párbeszédablakot (ugyanazt, mint amit a 22.5. ábrán láttunk).

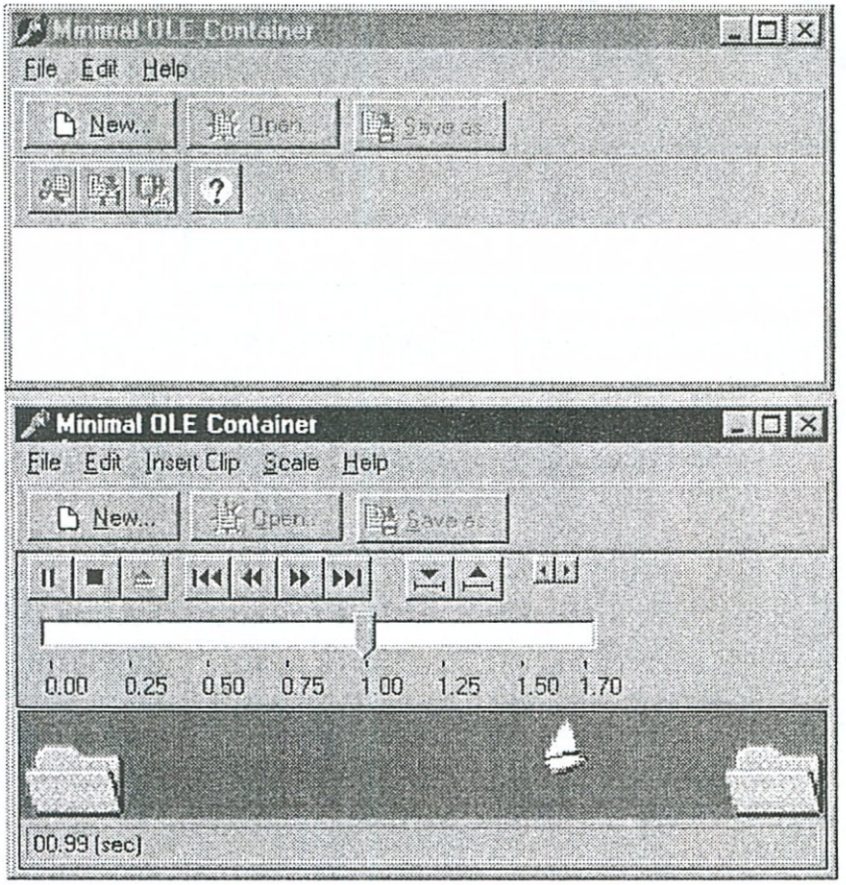
Az InsertObjectDialog metódus kirakja a párbeszédablakot, letörli az előző OLE objektumot, majd elvégzi az új objektum inicializációját, viszont nem aktiválja automatikusan azt. Emiatt aztán az alábbi kóddal szokták aktiválni az objektumot.

```
procedure TForm1.New1Click(Sender: TObject);
begin
  if OleContainer1.InsertObjectDialog then
    OleContainer1.DoVerb (OleContainer1.PrimaryVerb);
end;
```

Ha egyszer létrehoztunk egy új objektumot, akkor a DoVerb metódussal végrehajthatjuk az elsődleges verbjét. A program kirak egy eszköztárat is, amelyen bitkép gombok vannak. Elhelyeztem néhány TWinControl komponenst a formban, hogy a felhasználó kiválaszthassa őket, ezáltal érvénytelenítve az OleContainert. Hogy ez az eszköztár látható maradjon az OLE szerver működése alatt is, az eszköztár Locked tulajdonságát `True`-ra kell állítani.

A vizuális szerkesztés és az eszköztárak

Mint már említettem az OleCont2 példa akkor is megtartja eszköztárát, ha helyben aktiválunk egy OLE objektumot (ami persze lehet, hogy eleve lehetetlen, ha épp nem olyan szervert használunk). A most következő példában (OleCont3) újabb eszköztárat vezettem be, néhány új gombbal. Mivel ezúttal nem állítom be a Locked tulajdonságot, ezt az eszköztárat bizony felül fogja bírálni az aktív OLE szerver eszköztára. Ha a helyi szerkesztés egy olyan szerver-alkalmazást indít el, ami eszköztárat jelenít meg, akkor a szerver eszköztára felülbírálja a tároló eszköztárát. Ezt hivatott illusztrálni a 22.8. ábra alsó része.



22.8. ábra *Az OleCont3 példa második eszköztára (fent) lecserélődik a szerver eszköztárára (lent)*

Tipp

Az OLE tároló komponenst tegyük a panel komponensbe, és rendezzük mindkettőt a form klienshez tartozó területére, hogy az automatikus méretezés könnyen menjen.

A szabványos OLE párbeszédablakok

Az előző példában futásidőben hoztunk létre új OLE objektumot az InsertObjectDialog metódussal. Más módon is lehet OLE objektumot létrehozni: az egyik ilyen módszer az, hogy meghívjuk a PasteSpecialDialog nevű függvényt. Egy másik lehetséges megközelítés az, ha egyszerűen húzással hozunk létre OLE objektumot, ám ezt a módszert ebben a könyvben nem tárgyaljuk. A példa negyedik változatát (OleCont4) a Paste Special támogatással, és néhány további vágólap támogatással bővítettem (a Delphi Vágólap támogatását általánosabban a 30. fejezetben ismertetem, ezért a most következő kódnak nem magyarázom meg minden részletét).

A másik bővítés az OLE csatolások kezelése. Ez a művelet is elérhető Delphibe ágyazott szabványos párbeszédablakon keresztül. Több szabványos OLE párbeszédablak is létezik. Ezek közül van, amelyik közvetlenül elérhető Delphiből bizonyos speciális metódusok (mint például az InsertObjectDialog, az ObjectPropertiesDialog, és a PasteDialog) segítségével.

Az OleCont4 példa természetesen ugyanazt a formot használja, mint az előző verzió, csak éppen van egy új Object Properties nevű menüpont az Edit menüben. A most következő kód fut le ez utóbbi menüpont kiválasztásakor:

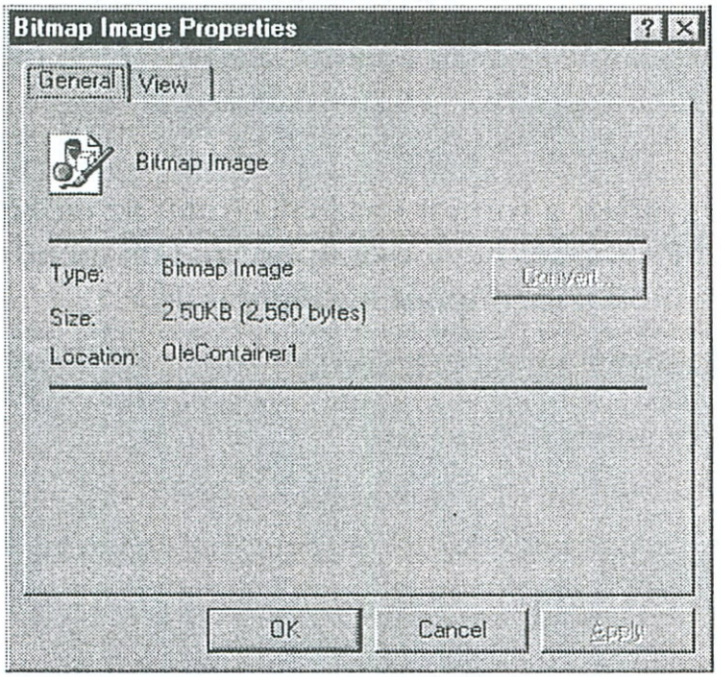
```
procedure TForm1.Object1Click(Sender: TObject);
begin
  OleContainer1.ObjectPropertiesDialog;
end;
```

A 22.9. ábrán láthatjuk a keletkező OLE párbeszédablakot. Természetesen ez a párbeszédablak a tárolóban lévő aktív OLE objektum jellegétől függően más és más lesz.

Továbbá a program most már kezeli a vágólapot is a kivágás, a másolás, a beillesztés, és az irányított beillesztés műveleteken keresztül. Most következzenek ezeknek a metódusok kódjai (ezek a kódok vezérlik egyébként a menüparancsokat, és az eszköztár gombjait is):

```
procedure TForm1.Paste1Click(Sender: TObject);
begin
  OleContainer1.Paste;
  UpdateUI;
end;

procedure TForm1.PasteSpecial1Click(Sender: TObject);
begin
  OleContainer1.PasteSpecialDialog;
  UpdateUI;
end;
```

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  OleContainer1.Copy;
  UpdateUI;
end;

procedure TForm1.Cut1Click(Sender: TObject);
begin
  OleContainer1.Copy;
  OleContainer1.Destroy;
  UpdateUI;
end;
```



22.9. ábra *Az OleCont4 példaprogramból elérhető a szabványos OLE Object Properties párbeszédablak*

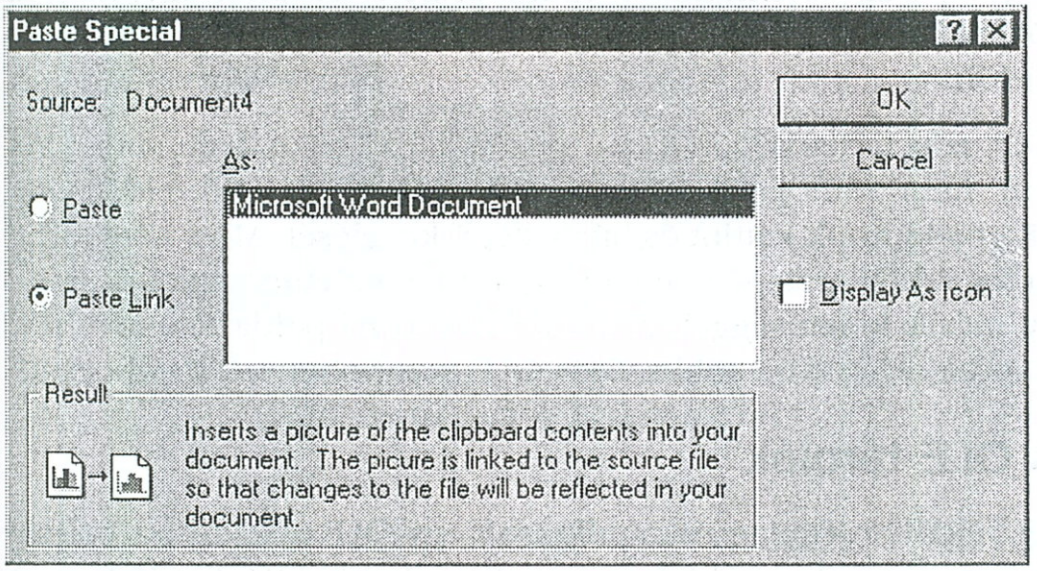
E négy metódus közül a legfontosabb a PasteSpecial1Click, ami egy újabb szabványos OLE párbeszédablakot jelenít meg. Ez utóbbit a 22.10. ábra szemlélteti.

Ezen kezelők mindegyike egy bizonyos UpdateUI eljárást hív meg (ezt teszi az Edit1Click metódus is, melynek feladata az Edit legördülő menü kiválasztásának kezelése):

```
procedure TForm1.UpdateUI;
begin
  Cut1.Enabled := OleContainer1.State <> osEmpty;
  Copy1.Enabled := OleContainer1.State <> osEmpty;
```

```
    Object1.Enabled := OleContainer1.State <> osEmpty;
    Paste1.Enabled := OleContainer1.CanPaste;
    PasteSpecial1.Enabled := OleContainer1.CanPaste;
    PasteSpeedButton.Enabled := Paste1.Enabled;
    CutSpeedButton.Enabled := Cut1.Enabled;
    CopySpeedButton.Enabled := Copy1.Enabled;
  end;
```

Ez a privát metódus az OleContainer komponens State tulajdonsága, és a CanPaste metódus által visszaadott érték alapján engedélyezi, illetve letiltja a menüpontokat és az eszköztár gombjait.



22.10. ábra *A szabványos OLE Paste Special párbeszédablak*

Objektumok betöltése fájlból és mentése fájlba

Végezzünk még egy utolsó módosítást ezen az OLE tárolón: legyen ez a fájlkezelés támogatása. Ez voltaképpen az egyik legegyszerűbb bővítés, amit csak el lehet képzelni, hiszen az OLE tároló komponensben eleve benne van a fájlkezelés támogatása. A új példaprogram (OleCont5) két szabványos OpenDialog és SaveDialog komponenst tartalmaz, a forráskódja pedig csupán néhány sorral bővül:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    OleContainer1.LoadFromFile (OpenDialog1.FileName);
  UpdateUI;
end;
```

```
procedure TForm1.SaveAs1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
    OleContainer1.SaveToFile (SaveDialog1.FileName);
end;
```

A kód ismét csak meghív két metódust a `TOleContainer` osztályból. Van itt egy másik metódus is – a CreateObjectFromFile – aminek teljesen más a jelentése, mint a LoadFromFile-nak. A LoadFromFile mindössze arra való, hogy a SaveToFile eljárással elmentett fájlokat betöltsük. Ebben a példában ez utóbbi metódusokat használtam, a fájloknak pedig a .DOL kiterjesztést adtam (ez a Demo OLE rövidítése kíván lenni). Ezzel szemben a CreateObjectFromFile arra használható, hogy új OLE objektumokat szúrjunk be a tárolóba úgy, hogy az adatot fájlból töltjük be. Ezt a fájlt a szerver hozza létre, a kiterjesztése pedig a szerver típusát jelöli.

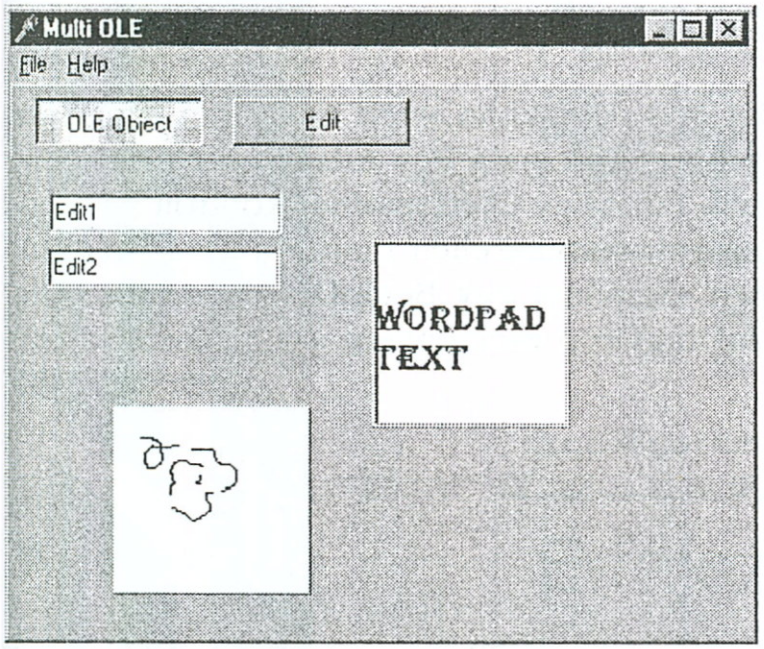
A program az OLE adta lehetőségek közül igen sokat kihasznál, de nem mindegyiket. Például nem támogatja objektumok fájlokból való előállítását, valamint az OLE objektumok kattint és húz jellegű kezelését. Most nem fogjuk ebbe az irányba továbbfejleszteni a példaprogramunkat, hanem inkább megismerkedünk egy másik lehetőséggel. A most következő példa lehetővé teszi különböző szerverekhez tartozó többszörös OLE komponensek létrehozását.

Többszörös OLE tárolók

Az összes eddigi példában egyszerre mindig csak egy OLE objektumot tudtunk megjeleníteni, ugyanis minden példában egyetlen állandó OLE tároló komponenst készítettünk. A következő példán keresztül megtanuljuk, hogy miként lehet ugyanabban a formban több OLE tároló komponenst kezelni (az olvasó akár össze is hasonlíthatja a mi példánk filozófiáját a Delphi saját MDI-s példájának filozófiájával).

A MultiOle példa OLE tárolókat és szerkesztőmezőket tud létrehozni és megjeleníteni. Akkor jönnek létre ezek a komponensek, ha a felhasználó duplán rákattint a form felületére, mégpedig pontosan a kattintás helyén. A program ezen kívül még mást is tud. Ha valamelyik szerkesztőmezőt vagy OLE tárolót elmozdítjuk a helyéről, attól még az OLE tárolók továbbra is egyetlen kattintással aktiválhatóak maradnak. Továbbá kiválaszthatunk egy OLE tárolót, és előugró menüjén keresztül meghívhatjuk a műveleteit. Az eszköztárban lévő két gomb állapotától függ, hogy a két komponens közül melyiket használjuk. Ezek teljesen úgy működnek, mint a rádiógombok, értékük pedig megegyezik a GroupIndex tulajdonság értékével. E két gomb OnClick eseménykezelője egyszerűen csak a form logikai OleObj mezőjét állítja be igaz illetve hamis értékre.

A FormDblClick metódus a forráskód egyik legfontosabb eleme. Ez hozza létre az új komponenst amikor a felhasználó duplán kattint a form felületére. A 22.11. ábrán látható, hogy mi történik, ha ez a kód már párszor lefutott.



22.11. ábra *A MultiOle példaprogrammal több OLE tárolót és szerkesztőmezőt is készíthetünk*

Voltaképpen van egy másik eseménykezelő metódus is, amivel az egér aktuális pozícióját lehet lekérdezni, és eltárolni a form X1, és Y1 mezőibe. A neve FormMouseDown. Erre azért van mindenképp szükség, mert az OnDblClick esemény semmiféle információt nem szolgáltat az egér pozíciójára vonatkozóan. A FormDblClick-nek két különálló része van: az egyik a szerkesztőmező, a másik az OLE tároló létrehozása. Itt most az OleContainer komponenshez tartozó kód következik:

```
procedure TForm1.FormDblClick(Sender: TObject);
var
  OleCont: TOleContainer;
  Pan: TPanel;
begin
  if OleObj then
  begin
    // create a panel to host the OLE container
    Pan := TPanel.Create (self);
    Pan.Parent := Self;
    Pan.SetBounds (X1, Y1, 100, 100);
    // create an OLE container, with unique name
    OleCont := TOleContainer.Create (self);
    OleCont.Parent := Pan;
```

```
        OleCont.Align := alClient;
        Inc (OleCount);
        OleCont.Name := 'OleContainer'+IntToStr (OleCount);
        // insert the actual object
        OleCont.InsertObjectDialog;
      end
      else ...
```

Láttunk már ehhez hasonló kódot más példákban, ezért most ezt nem is részletezem. Van viszont ennek a programnak még egy tulajdonsága. Lehetővé teszi, hogy a felhasználó bármelyik Edit vagy OleContainer kontrolt a formon belül akárhová mozgassa. Ehhez csak annyit kellett tenni, hogy az irányítók létrehozásakor az OnMouseDown eseményt egy BeginDrag metódust meghívó kezelőre állítjuk. A form OnDragDrop eseményére a következő kezelőt írtam:

```
procedure TForm1.FormDragDrop(
  Sender, Source: TObject; X, Y: Integer);
var
  Control: TControl;
begin
  if Source is TOleContainer then
    // move the panel hosting the control
    Control :=  (Source as TControl).Parent
  else
    Control := Source as TControl;
  Control.Left := X;
  Control.Top := Y;
end;
```

Szokás szerint e példának teljes forráskódja is megtalálható a CD mellékleten.

Hogyan tovább?

Ebben a fejezetben a Microsoft OLE technológiájának két implementációjával foglalkoztunk. Az egyik az OLE Automation, a másik pedig az OLE dokumentumok. Bizonyára feltűnt, hogy a Delphi mindkettőt nagymértékben támogatja. Már a Delphi 2 is támogatta őket, csak éppen másképpen, és kevésbé hatékonyan.

Van valami, ami viszont teljesen új a Delphi 3-ban. Ez az ActiveX kontrolok és ActiveFormok vizuális kezelése. A következőkben ezzel ismerkedünk meg. Ez a két dolog ugyanazon a technológián alapul (az ActiveX-en), viszont különböző célokra szánták őket. Az ActiveX kontrolok olyan vezérlők, amelyeket a Delphin kívül más fejlesztői környezetben is lehet használni. Az ActiveFormok pedig olyan Delphi formok, amiket internetes dokumentumokba is bele lehet ágyazni. Tulajdonképpen e két technológia közötti határvonal nem ennyire éles, ám ezek a legtipikusabb felhasználások, amint azt a most következő két fejezetben látni is fogjuk.

23. FEJEZET

ActiveX kontrolok használata és készítése

E fejezet tartalma:

- AcitveX vezérlők bemutatása
- ActiveX vezérlők használata Delphiben
- ActiveX vezérlő írása

ActiveX kontrolok használata és készítése

A Microsoft Visual Basic volt az első fejlesztői környezet, amely alkalmazta a szoftverkomponensek használatát és terjesztését. Az újra felhasználható komponensekkel való fejlesztés ötlete viszont már előbb is megjelent, magában az objektum-orientált programozás alapjaiban is megtalálhatjuk. Az OOP nyelvek nem tudták biztosítani az ígért újrafelhasználhatóságot, talán piaci és szabványosítási problémák miatt. Visual Basic ellenben nem használja ki az objektum-orientált programozás előnyeit, de átvesz abból ötleteket a definíciókkal és a rendszerbe való integrálással kapcsolatban. Az első szabvány, amelyet a Visual Basic támogatott a VBX, ennek 16-bites változata a 16-bites Delphi-ben is felhasználható. A 32-bites rendszerek előretörésével a Microsoft a VBX helyett egy új, erőteljesebb, és még inkább nyitott szabványt vezetett be, amely az ActiveX nevet kapta.

Megjegyzés

Az ActiveX vezérlőket OLE vezérlőknek (OCX) is szokták hívni. Az új név inkább egy új piaci stratégiát sugall a Microsoft részéről, mint technikai újítást. Az ActiveX tekinthető az OCX technika részbeni kiterjesztésének is, ezért nem meglepő, hogy az ActiveX vezérlőket továbbra is sokszor OCX kiterjesztésű fájlokban tároljuk.

ActiveX vezérlők bemutatása

Nagy vonalakban ezek az ActiveX vezérlők nem sokban különböznek bármely Windows-beli, Delphi-béli vagy Visual Basic-beli társaiktól, amelyek mindig valamilyen ablakként határozták meg a vezérlőket, amelyhez tartozik egy kód, ami viselkedését meghatározza. A legnagyobb különbség a vezérlők különböző fajtái között az, hogy azok milyen kölcsönhatásban vannak az alkalmazás többi részével. Windows-os vezérlők üzenetalapú felületet használnak, VBX vezérlők tulajdonságokat és eseményeket, OLE objektumok tulajdonságokat és metódusokat, az ActiveX vezérlők pedig tulajdonságokat, metódusokat és eseményeket. Az utolsó három elem a Delphi saját komponenseiben is felfedezhető.

Az OLE szóhasználatával élve az ActiveX nem más, mint „összetett dokumentumokból álló objektum folyamatbeli DLL szerverként történő megvalósítása, amely támogatja az OLE Automationt, a vizuális szerkesztést és a fordított aktiválást". Most lássuk, hogy ez valójában mit takar...

Az ActiveX használata az OLE szerverekhez hasonló programozási szemléletet igényel. Azt már tudjuk az előző fejezetből, hogy az OLE szerver nem más, mint egy OLE objektum, amit egy OLE dokumentumba beágyazunk. A különbség az OLE szerver és az ActiveX vezérlő között az, hogy az OLE szervert három különböző módon valósíthatjuk meg:

- Egy önálló alkalmazásként (pl. Microsoft Excel).
- Olyan futtatható fájlokként, amelyek önmagukban nem működőképesek, csak egy szerveralkalmazásból meghívva fejtik ki hatásukat, angolul ezeket out-of-process szervereknek nevezzük (pl. Microsoft Graph és a hozzá hasonló alkalmazások).
- DLL fájlokként, amelyek ugyanazt a memóriaterületet használják, mint az őket használó programok. Az ilyen típusú szervereket in-process névvel illetjük.

Az ActiveX vezérlők csak a legutóbbi technikával valósíthatók meg, amely egyben a leggyorsabb is. Továbbá az ActiveX vezérlők OLE szerverek is egyben, ami azt jelenti, hogy el lehet érni azok tulajdonságait, és meg lehet hívni metódusaikat. Az OLE felületéből viszont hiányzanak az események, amelyek az ActiveX-nél már megtalálhatók, és ez teszi az ActiveX-et a Delphi komponenseihez hasonlóvá.

Az ActiveX vezérlők mindig láthatók az őket tartalmazó alkalmazásablakokban, és közvetlenül kommunikálhatunk is velük. Ezt a tulajdonságot nevezzük vizuális szerkesztésnek, vagy „helyi" aktiválásnak. Egy szimpla kattintás aktiválja a vezérlőt, nem úgy, mint az OLE dokumentumoknál, amelyeken duplán kell kattintani, továbbá e vezérlők mindig aktívak amikor láthatóak. Ez utóbbi tulajdonságot nevezzük fordított aktiválásnak.

Ahogy már az korábban említettük, egy ActiveX vezérlőnek lehetnek tulajdonságai, metódusai és eseményei. A tulajdonságokat állapotok jelzésére használjuk, de metódusokat is hívhatnak (ez különösen igaz olyan ActiveX vezérlőkre, amelyeket VBX komponensekből hoztak létre, mivel VBX esetén nem volt más lehetőség metódusok hívására, csak tulajdonságok állításán keresztül). Tulajdonságok vonatkozhatnak származtatott értékekre, tömbökre, alobjektumokra, stb. A tulajdonságok továbbá lehetnek dinamikusak (csak olvashatók a Delphi terminológia szerint), vagy statikusak.

ActiveX esetén a tulajdonságok több különböző csoportra osztottak: csoporttulajdonságok (stock properties), amelyek minden vezérlőhöz nélkülözhetetlenek azok megvalósításához, átlátszó tulajdonságok (ambient properties), amelyek a tartalmazó objektumról (azaz a konténerről) adnak információt (pl. ParentColor vagy ParentFont a Delphi-ben), kiterjesztett tulajdonságok (extended properties), amelyeket a konténer kezel, mint például az objektum helyzete, és végül a testre szabott tulajdonságok (custom properties), amik bármit jelenthetnek.

Az események egér kattintására, billentyűlenyomásra, a komponens aktivizálódására jönnek létre ugyanúgy, ahogy azt eddig megszoktuk. A Delphi és az ActiveX eseményei és metódusai között nincs is semmi lényeges eltérés.

ActiveX vezérlők kontra Delphi komponensek

Mielőtt bemutatnánk, hogyan kell ActiveX vezérlőt írni Delphi alatt, ismételjük át még egyszer a két vezérlőtípus közötti különbséget. Az ActiveX vezérlők DLL alapúak, ami azt jelenti, hogy ha használni akarjuk őket, akkor a kódjukat tartalmazó OCX fájlt a programmal együtt kell terjeszteni. Delphi komponensek esetén azok kódját a program futtatható fájljához statikusan hozzáfűzhetjük, vagy dinamikus alkönyvtárakban hozzákapcsoljuk, tetszésünk szerint.

Mivel több fájlban tároljuk programunkat, lehetővé válik, hogy ugyanazt a kódrészletet több program is megossza egymás között, ahogy ezt DLL-ek esetén is teszik. Így ha két alkalmazás ugyanazt a vezérlőt vagy csomagot használja, azt elég egyszer tárolni a merevlemezen és egyszer betölteni a memóriába, ezáltal helyet takaríthatunk meg. Ennek a módszernek a hátránya az, hogy ha a két alkalmazás két különböző változatot használ ugyanabból az ActiveX vezérlőből, akkor kompatibilitási problémák léphetnek fel. Továbbá ha minden szükséges kódrészlet a futtatható fájlban található, alkalmazásunk telepítése leegyszerűsödik, és biztonságosabbá válik.

Ezek után mi a Delphi komponensek hátránya? A legnagyobb probléma nem az, hogy kevesebb Delphi komponens van, mint ActiveX vezérlő, hanem az, hogy ha van egy Delphi komponensünk, akkor azt csak Delphi-ben és Borland C++ Builder-ben tudjuk majd használni, míg egy ActiveX vezérlőt több különböző cég különböző szoftvereiben felhasználhatjuk. Abban az esetben, ha főleg Delphiben fejlesztünk, és egy adott komponens mindkét változata rendelkezésünkre áll, célszerű a Delphi komponens választani, mert annak használata könnyebb, és jobban illeszkedik a rendszerbe. Az eredeti Delphi komponensek feltehetően jobban lesznek dokumentálva (főleg a Pascal nyelv szempontjából), és jobban kihasználják a Delphi és az Object Pascal nyelvet, mint egy általánosabb ActiveX vezérlő, amelynek alapja a C és a C++.

ActiveX vezérlők használata Delphiben

Delphi-ben is találunk pár ActiveX vezérlőt, de beszerezhetünk más cégek által kifejlesztetteket, amelyeket könnyen lehet telepíteni a Delphi környezetbe (valójában a Delphi 3 alapértelmezett ActiveX vezérlőit is külső cég készítette, és nem a Borland). Mivel az ActiveX vezérlők a Delphi Standard verziójában nem találhatók, először szeretnénk az ActiveX vezérlők általános működését bemutatni, és csak utána mutatni pár egyszerű példát.

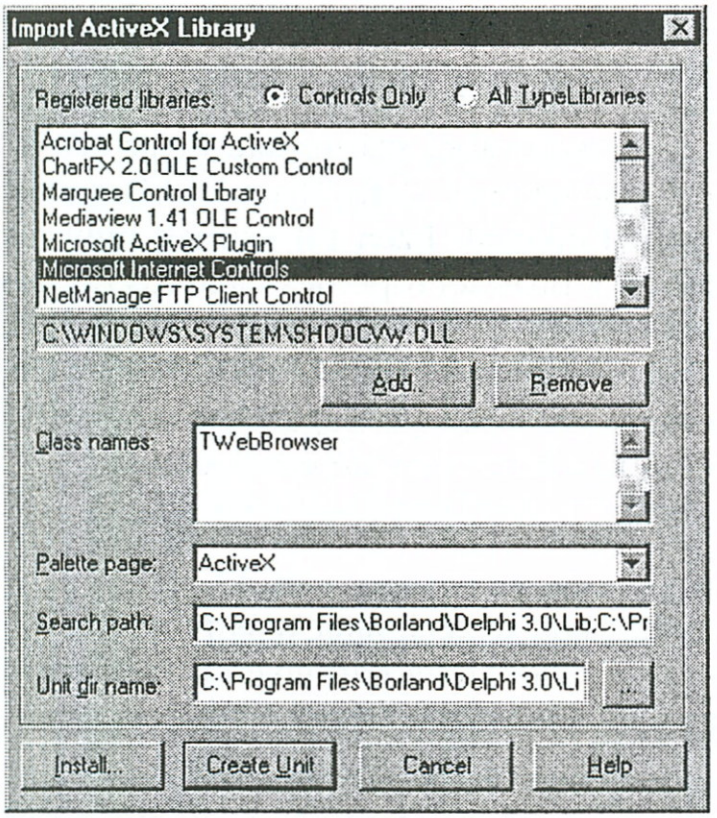
ActiveX vezérlők telepítése

Ha már telepítettünk ActiveX vezérlőket tartalmazó könyvtárat számítógépünkre, akkor utána a Delphiben is telepíteni kell azokat, hogy megjelenjenek a fejlesztői környezet megfelelő helyén. Az ActiveX vezérlőkönyvtár az OCX fájl pontos megnevezése, ami valójában több ActiveX vezérlőt is tartalmazhat.

Delphi alatt a telepítés meglehetősen egyszerű. Válasszuk ki a **Components** menüből az **Install ActiveX library** almenüt. Ez megnyitja az Import ActiveX Library párbeszédablakot (23.1. ábra). Itt láthatók a Windows alá eddig telepített ActiveX vezérlőkönyvtárak. Ha ezek közül egyet kiválasztunk, akkor a könyvtár tartalma megjelenik a képernyőn. Láthatjuk a benne lévő vezérlőket, és azoknak a Delphi által javasolt neveit. Ha a fentieket elfogadjuk, kattintsunk a **Create Unit** gombra, amelynek hatására a Delphi elkészíti a vezérlők Pascal kódját. Ezek után az **Install** gombra kattintva az adott egység telepítése befejeződik, és a vezérlő megjelenik a komponenspalettán.

A Delphi által generált forráskódban találhatjuk az osztálydefiníciókat a könyvtárban található minden ActiveX vezérlőre. Minden vezérlőhöz tartozik egy `interface` és egy `dispinterface` típusú deklaráció. Az alábbi példa a WebBrowser-hez tartozik, amit a 23.1. ábrán választottunk ki. A Delphi által generált kód részlete látható itt:

```
type
  IWebBrowser = interface(IDispatch)
    ['{EAB22AC1-30C1-11CF-A7EB-0000C05BAE0B}']
    procedure GoBack; safecall;
    procedure GoForward; safecall;
    procedure GoHome; safecall;
    ...

  DWebBrowser = dispinterface
    ['{EAB22AC1-30C1-11CF-A7EB-0000C05BAE0B}']
    procedure GoBack; dispid 100;
    procedure GoForward; dispid 101;
    procedure GoHome; dispid 102;
    ...

  TWebBrowser = class(TOleControl)
  private
    FIntf: IWebBrowser;
  public
    procedure GoBack;
    procedure GoForward;
    procedure GoHome;
```



23.1. ábra *Az ActiveX vezérlő telepítését végző Delphi párbeszédablak*

Az alapvető elképzelés szerint a Delphi deklarál egy interfészt, majd a `TOleControl` osztályból létrehoz egy alosztályt. Az új osztály a `TWebBrowser` nevet viseli, és nem implementálja az új interfészt, hanem lokális privát mezőként tartalmaz egy interfész objektumot. Így a `TWebBrowser` osztály tartalmazza mindazokat a tulajdonságokat, metódusokat és eseményeket, mint amiket az ActiveX vezérlő, és párat, amelyet a `TOleControls` osztálytól örökölt.

A Delphi minden esetben megpróbálja megtartani a tulajdonságok eredeti neveit, de ez nem mindig lehetséges. Amikor a Pascal fájlok létrehozása történik, vele egy időben a fordító egy segédfájlt is létrehoz, amiben jegyzi azokat az eseteket, amikor egy adott nevet meg kellett változtatnia, ahogy azt a következő példa is mutatja.

```
{Warning: 'Type' is a reserved word.
IWebBrowser.Type changed to Type_}
```

ActiveX vezérlők telepítésekor kövessünk a fent bemutatott lépéseket. Ezek után tanulmányozzuk (esetleg nyomtassuk ki) a vezérlő Object Pascal forráskódját, és nézzük át, hogy az a Delphi alatt miként lett definiálva. A fájl elején megtalálhatjuk a konstansok listáját is.

A TOleControl osztály

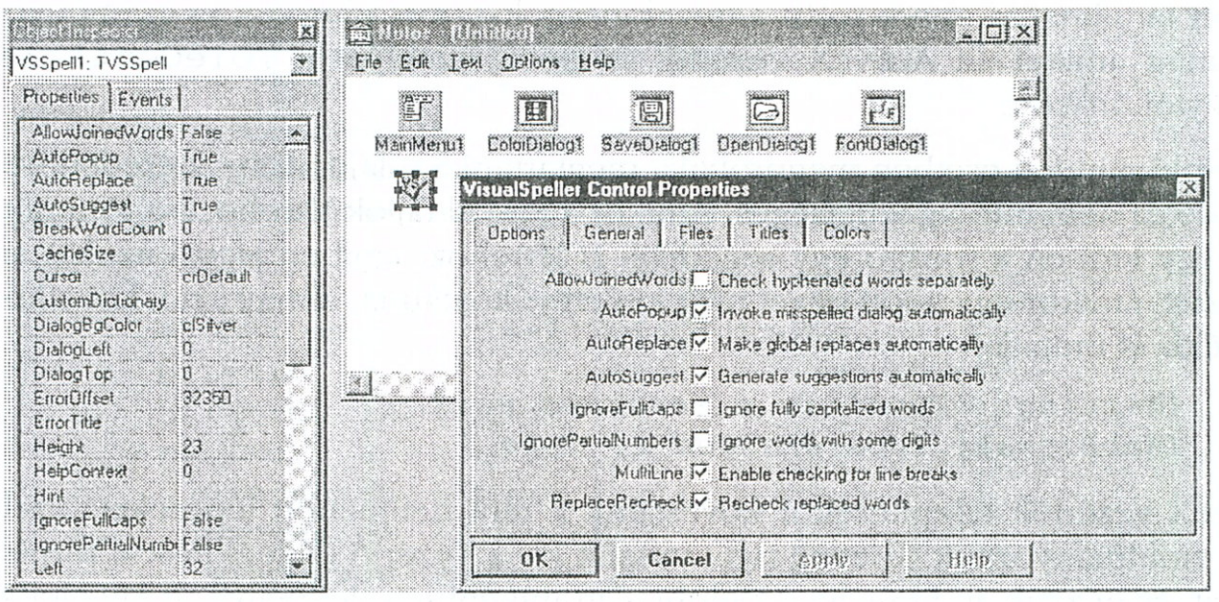
Minden ActiveX vezérlő telepítésekor a Delphi definiál egy új Object Pascal osztályt, amelyet a VCL `TOleControl` osztályából származtat. Ez az osztály magában foglalja az ActiveX alapvető viselkedését, definiálja a kódját, amivel eléri és kezeli tulajdonságait és egyéb funkcióit. Ezen túl vannak publikus metódusai és tulajdonságai, mint például a BrowseProperties metódus vagy az OleObject tulajdonság. Ezekkel lehet elérni a belső OLE objektumot, ami az ActiveX vezérlőt definiálja. Ez lehetővé teszi a vezérlő objektum közvetlen elérését, ami viszont bonyolultabb program írását igényli. Legtöbbször a vezérlő Delphin keresztüli használata egyszerűbb a közvetlen elérésnél.

ActiveX vezérlők használata

Miután az ActiveX vezérlőt telepítettük (akár kézzel, akár más programmal), ugyanúgy hozzáadhatjuk az alkalmazásunkhoz, mint bármilyen másik Delphi komponenst. Ezek után beállíthatjuk a tulajdonságait, kezelhetjük eseményeit, ahogy azt megszoktuk.

Első példánk legyen a Spell című program, ami a VSSpell ActiveX vezérlőt használja, ami a Delphi 3 Professional kiadásának komponensei között található. Ebben egy régebbi, Jegyzettömb típusú alkalmazást írunk át úgy, hogy azt egy helyesírás-ellenőrzés funkcióval látjuk el. Így néz ki a komponens forráskódja:

```
object VSSpell1: TVSSpell
  ControlData = {00000200010101000000000.....}
end
```



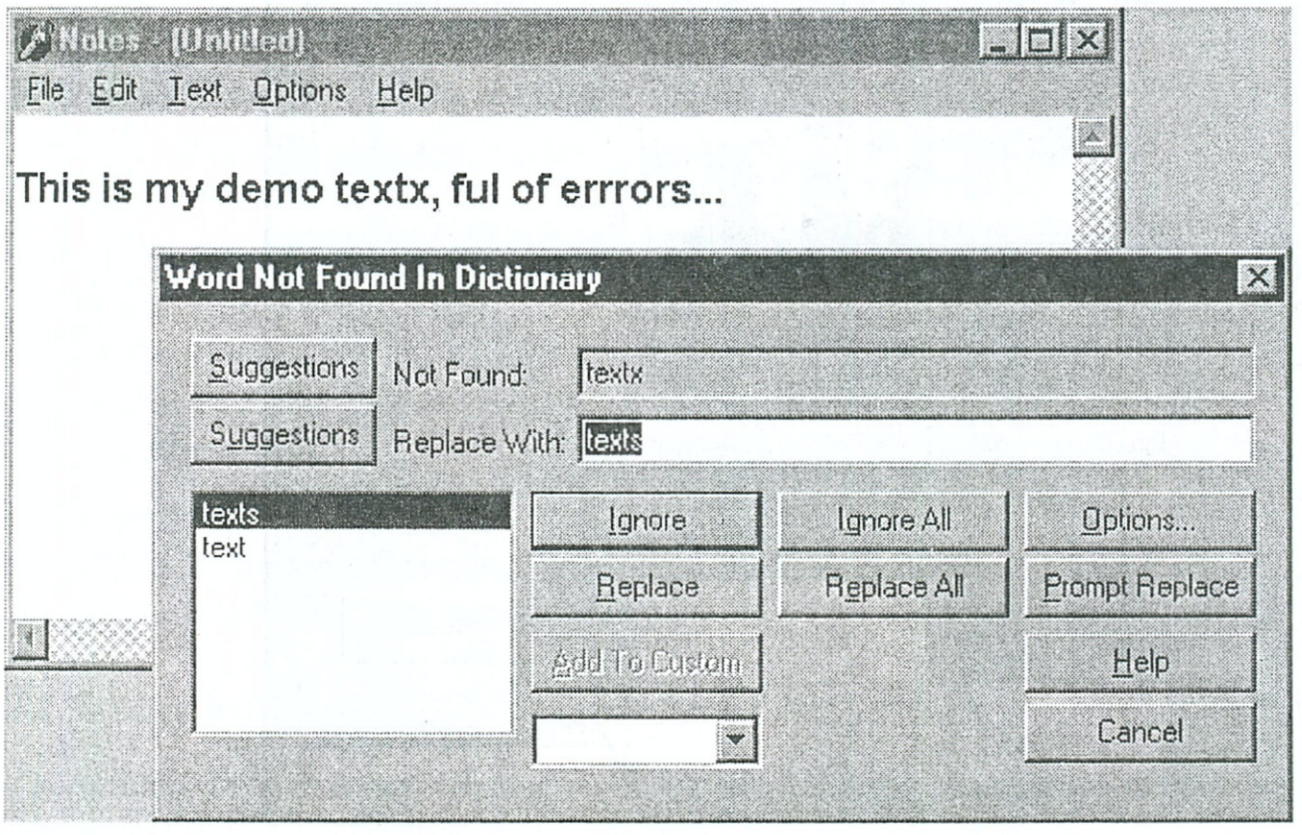
23.2. ábra *Az Object Inspector és a vezérlő tulajdonságszerkesztője: e két módszer szolgál az ActiveX tulajdonságának szerkesztésére*

Ugyan a fenti számokat nehéz értelmezni, mégis láthatjuk a ControlData tulajdonságon keresztül, hogy az ActiveX vezérlő valóban egy beágyazott objektum. Ez a numerikus adat a vezérlő beállításait tartalmazza, amelyek vagy az Object Inspectoron keresztül, vagy a vezérlőre duplán kattintva annak szerkesztőjén keresztül változtathatjuk meg, ahogy ez a 23.2. ábra is mutatja.

A Spell példában, hogy az új funkciót el lehessen érni, a menühöz hozzá kellett adni egy új bejegyzéssel, és a menüpont OnClick eseményét is kezelni kell, ami a helyesírás-ellenőrzést indítja el.

```
procedure TNotesForm.Spelltext1Click(Sender: TObject);
begin
  VSSpell1.CheckText := Memo1.Text;
  if VSSpell1.ResultCode = 0 then
    // the spelling was properly executed
    Memo1.Text := VSSpell1.Text;
end;
```

A program futása alatt válasszuk ki az **Options** menüből a **Spell** menüpontot. Ekkor az ActiveX megjelenít egy párbeszédablakot, amiben a helyesírás-ellenőrzési funkciókat el tudjuk végezni. Ez látható a 23.3. ábrán.



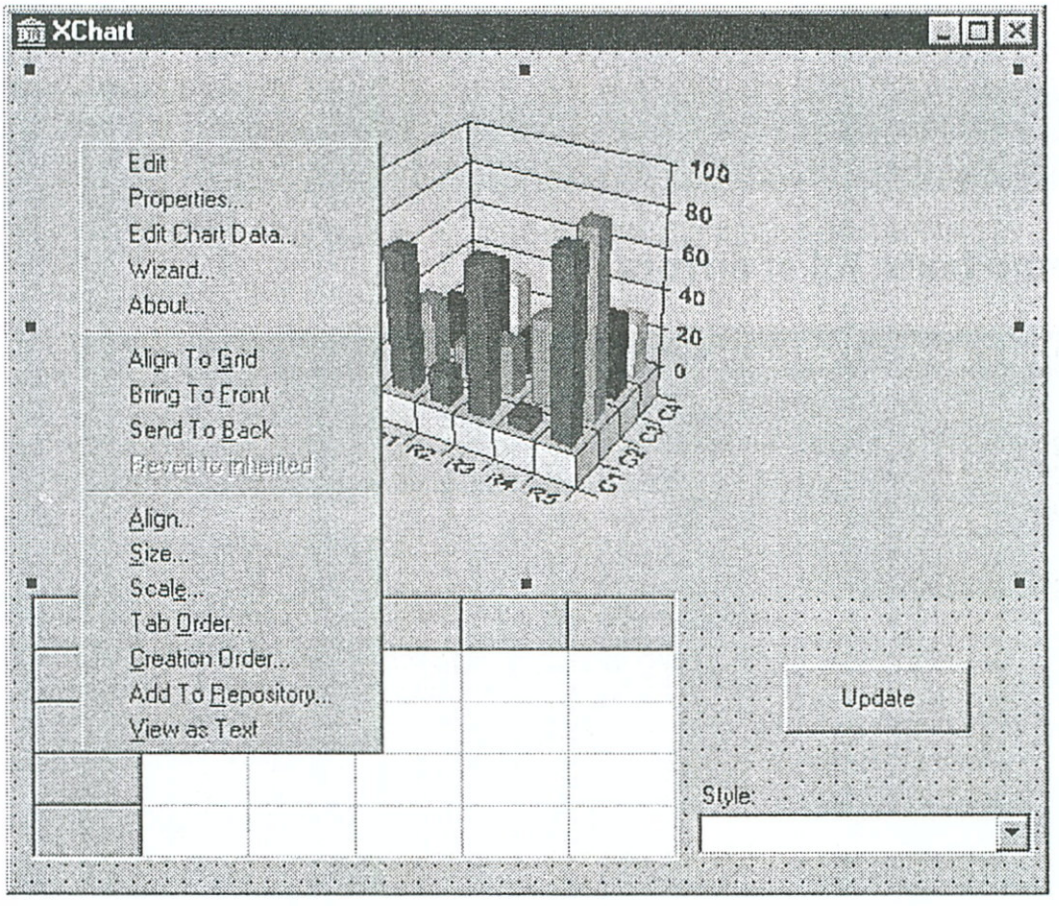
23.3. ábra *A VSSpeller ActiveX vezérlő működés közben*

Grafikonok készítése

A 11. fejezetben már áttekintettük a TeeChart komponenst, de a Delphi tartalmaz három különböző ActiveX vezérlőt grafikonok készítésére: a Software FX cég ChartFX termékét, továbbá a Bits Per Second Graph Custom Control és a Visual Components First Impression vezérlőit. Ezek igen fejlett komponensek (annak ellenére, hogy a Delphiben lévő verziók képességeit korlátozták), amelyek egyikén szeretnénk bemutatni, milyen egyszerű is vizuális ActiveX-eket használni.

Az egyetlen nehézséget a vezérlők Súgó fájljainak megtalálása okozhatja, amik a `Delphi 3.0\OCX` alkönyvtárban találhatók, de mivel főleg Visual Basic programozók számára készültek, kissé zavaró Delphi alatt használni őket.

A First Impression komponenst a XChart nevű programon keresztül mutatjuk be, ami nagyon hasonlít a Delphi TeeChart komponensét bemutató példaprogramra. Ahogy az a 23.4. ábrán is látszik, a képernyő tetején található a grafikon, alatta egy táblázat és egy gomb, amivel a numerikus adatokat lehet a táblázatból a grafikonra másolni.



23.4. ábra *Az XChart példa formja szerkesztési időben, az ActiveX vezérlő menüjével*

A kombinált lista segítségével tudjuk kiválasztani a grafikon típusát. Ahogy az eredeti példában, a program lényege itt is az UpDateButtonClick metódusban foglal helyet. Ahhoz, hogy a grafikonra adatot vigyünk az ActiveX vezérlőn a Row (sor) és Column (oszlop) tulajdonságok értékeit be kell állítani, ezek jelzik azt az értéket, amit változtatni akarunk a Data tulajdonsággal.

```
procedure TForm1.UpdateButtonClick(Sender: TObject);
var
  I, J: Integer;
begin
  for I := 1 to 5 do
  begin
    VtChart1.Row := I;
    for J := 1 to 4 do
    begin
      VtChart1.Column := J;
      VtChart1.Data := StringGrid1.Cells [I, J];
    end;
  end;
end;
```

ActiveX vezérlő írása

A Delphi 3-ban lévő ActiveX vezérlők használata mellett könnyen írhatunk saját vezérlőket is. Az ActiveX vezérlő kódját elkészíthetjük magunk is, ha létrehozzuk a szükséges OLE interfészeket, de sokkal könnyebb az ActiveX Control varázslót használni, ami ezt mind elvégzi helyettünk, megspórolva számunkra sok időt és vesződséget.

Új ActiveX írásakor először egy létező VCL komponensből indulunk ki, amely mindenképpen a `TWinControl` leszármazottja kell legyen, és e köré építjük saját ActiveX vezérlőnket. E közben a Delphi a vezérlőhöz hozzáad egy típuskönyvtárat is. Ez az eljárás pontosan fordítottja annak, ami az ActiveX Delphi környezetbe való implementálásakor végeztünk.

Következő lépésben elkészítjük a tulajdonságlapot, amely a tulajdonság szerkesztőhöz hasonlóan a vezérlő kezdeti értékeinek beállítására szolgál. A 23.2. ábra mutatja a VCSpell tulajdonságlapját.

Az Arrow komponens átírása ActiveX vezérlővé

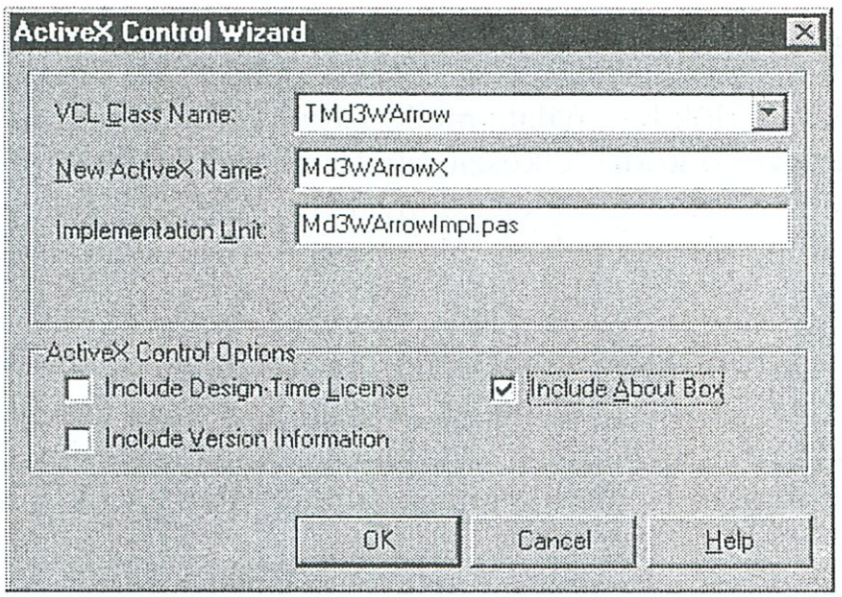
Az ActiveX fejlesztésének következő fázisaként a 17. fejezetben megismert Arrow komponenst kívánjuk átírni ActiveX-szé. Sajnos ezt nem tehetjük meg automatikusan, mivel az Arrow komponens a `TGraphicsControl` leszármazottja, viszont a grafikus vezérlő ablak-alapúvá alakítása általában járható út. Ezért a szülőosztály típusát lecseréltük `TCustomControl`-ra:

```
type
  TMd3WArrow = class(TCustomControl)
  ...
```

A `TWinControl` osztály minimális a grafikus támogatással rendelkezik, de ennek `TCustomControl` alosztálya szinte megegyezik a `TGraphicsControl` osztállyal. A különbség mindössze annyi, hogy a `TCustomControl`-nak van ablakleírója.

Miután telepítettük az új komponenst a Delphi alá, hozzákezdhetünk egy új példa készítéséhez. Először a **File** menüben válasszuk ki a **New** pontot, majd a fülek közül az **ActiveX**-et, és végül **ActiveX library**. Ezek után a Delphi elkészíti a DLL vázát, ahogy azt már a 21. fejezetben láttunk. Mentsük el XArrow néven egy ugyanolyan nevű mappába, ahogy szoktuk.

Ezek után használjuk az ActiveX Control varázslót, amit az Object Respository **ActiveX** lapján találunk. Ebben a varázslóban (a 23.5. ábra alapján) válasszuk ki a VCL osztályt, adjuk meg a nevet a szerkesztőmezőkben, kattintsunk az OK billentyűre, és a Delphi elkészíti az ActiveX vezérlő teljes forráskódját.

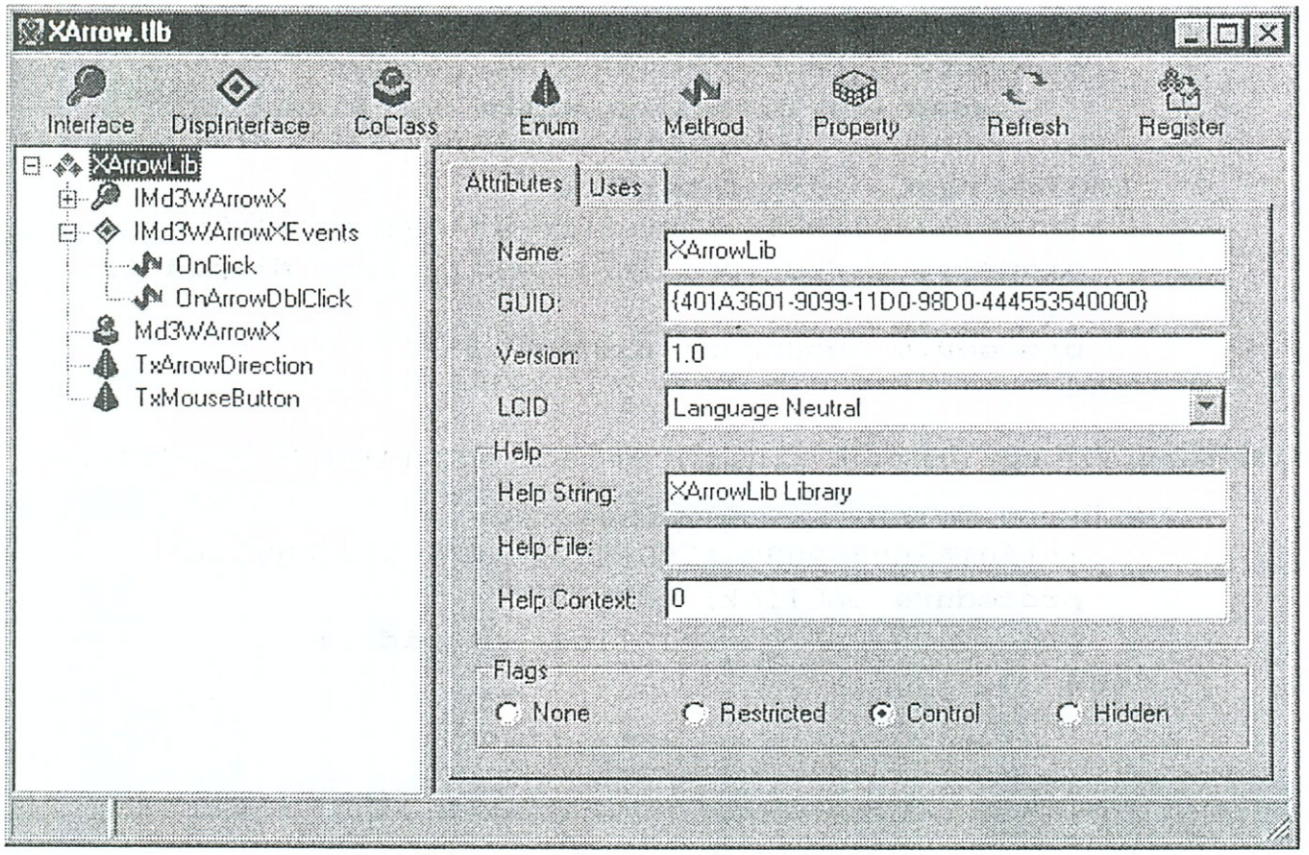


23.5. ábra *A Delphi ActiveX Control varázslója*

Az ActiveX Control varázsló ablak alján lévő három jelölőnégyzet használata nem feltétlenül egyértelmű, ezért ezek használatáról essen most pár szó. Ha az **Include Design-time Licence**-t bekapcsoljuk, akkor más felhasználó nem tudja a vezérlőt használni, hacsak nincs meg neki az ehhez szükséges regisztrációs szám. Ebben az esetben az ActiveX Control varázsló létrehozza ezt a regisztrációs számot, amelyet egy LIC kiterjesztésű fájlban tárol. A második

jelölőnégyzet lehetővé teszi, hogy az OCX fájlba az adott ActiveX-ről verzió-számot tárolhassunk el. Ennek használatáról bővebben a 27. fejezetben lesz szó. Ha a harmadik jelölőnégyzetet bekapcsoljuk, akkor az ActiveX Control varázsló automatikusan ad egy Névjegy ablakot az adott vezérlőhöz. Ebben a példában csak az utolsó jelölőnégyzetet kapcsoltuk be, hogy a Névjegy automatikusan létrejöjjön, igaz ez egy kis plusz munkát igényel, mégpedig az ablak címkéjének, és a többi szövegnek a megadását.

Fontos még, hogy megnézzük az ActiveX Control varázsló által generált kódot, ezen belül is a típuskönyvtárt. A 23.6. ábrán látható is az Arrow komponensből készített típuskönyvtár. Ebből a típuskönyvtárból a Delphi létrehoz továbbá egy import fájlt, amiben megtalálhatók az `interface` és `dispinterface` definíciók, típusok és egyéb konstansok.



23.6. ábra *A típuskönyvtár-szerkesztő az általunk készített ActiveX vezérlő típuskönyvtárával*

Itt láthatjuk az XArrowLib.pas import fájlt. A fájl első része tartalmazza GUID részeket, a könyvtár egészére, a vezérlőre, továbbá konstansokat, amelyek a Delphi vezérlő különböző tulajdonságainak értékeit adják meg, ahogy azt a következő példa is mutatja:

```
{ TxArrowDirection }
const
  adUp = 0;
  adDown = 1;
  adLeft = 2;
  adRight = 3;
```

A deklarációk lényege az IMd3WArrrowX `interface`, és a hozzá tartozó DMd3WArrowX `dispinterface` részek. Ezekből láthatunk két rövid példát:

```
type
  IMd3WArrowX = interface(IDispatch)
    ['{401A3602-9099-11D0-98D0-444553540000}']
    function Get_Direction: TxArrowDirection; safecall;
    procedure Set_Direction(
      Value: TxArrowDirection); safecall;
    ...
    procedure AboutBox; safecall;
    property Direction: TxArrowDirection
        read Get_Direction write Set_Direction;
    ...
  DMd3WArrowX = dispinterface
    ['{401A3602-9099-11D0-98D0-444553540000}']
    property Direction: TxArrowDirection dispid 1;
    ...
    procedure AboutBox; dispid -552;
  end;
```

Valójában van egy második `dispinterface` rész a vezérlő eseményeire:

```
IMd3WArrowXEvents = dispinterface
    ['{401A3603-9099-11D0-98D0-444553540000}']
    procedure OnClick; dispid 1;
    procedure OnArrowDblClick; dispid 2;
  end;
```

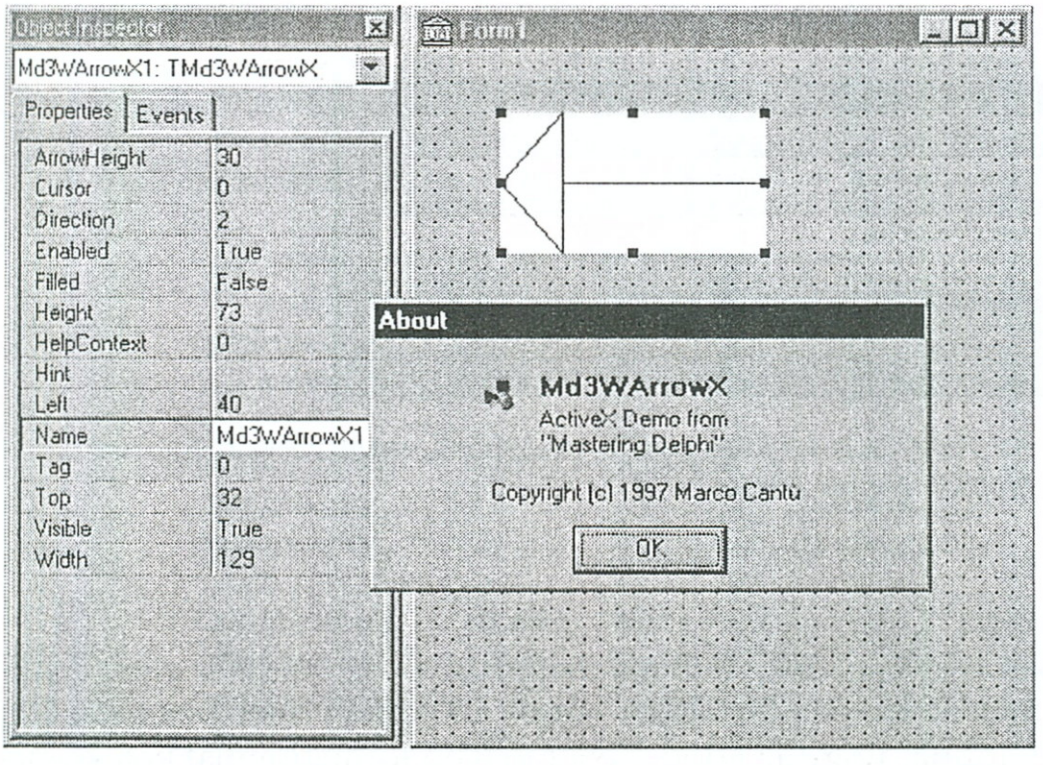
Megjegyzés

Az import egység vége tartalmazza a TMd3WArrowX deklarációját. Ezt az `TOleControl`-ból származtatott osztályt használjuk a Delphibe való telepítéskor, ahogy azt a fejezet elején láttuk. Ez az osztály nem szükséges a vezérlő elkészítéséhez, csak a Delphibe való telepítéséhez kell. Az ActiveX szerver által használt név megegyezik az osztály nevével, viszont az implementációja különbözik attól.

A kód többi része a főegységben van, ami esetünkben a XArImpl.pas fájl. Itt található az ActiveX szerver deklarációja, a `TMd3WArrowX`, amit a `TActiveXControl`-tól örököl, és az `IMd3WArrowX` interfész. Ennek a deklarációnak egy rövid részét láthatjuk itt:

```
type
  TMd3WArrowX = class(TActiveXControl, IMd3WArrowX)
  private
    FDelphiControl: TMd3WArrow;
    FEvents: IMd3WArrowXEvents;
    ...
  protected
    procedure InitializeControl; override;
    procedure DefinePropertyPages(
      DefinePropertyPage: TDefinePropertyPage);
override;
    function Get_ArrowHeight: Integer; safecall;
    procedure Set_ArrowHeight(Value: Integer);
safecall;
    procedure AboutBox; safecall;
    ...
```

Mielőtt tovább dolgoznánk, már láthatjuk is, hogy működik a vezérlő. Először le kell fordítani az ActiveX könyvtárat, és regisztrálni kell Delphiben, amit a **Run** menü **Register ActiveX Server** menüpontjával érhetünk el. Ezek után a telepítést végezhetjük úgy, mint eddig, de van erre egy gyorsabb megoldás is.



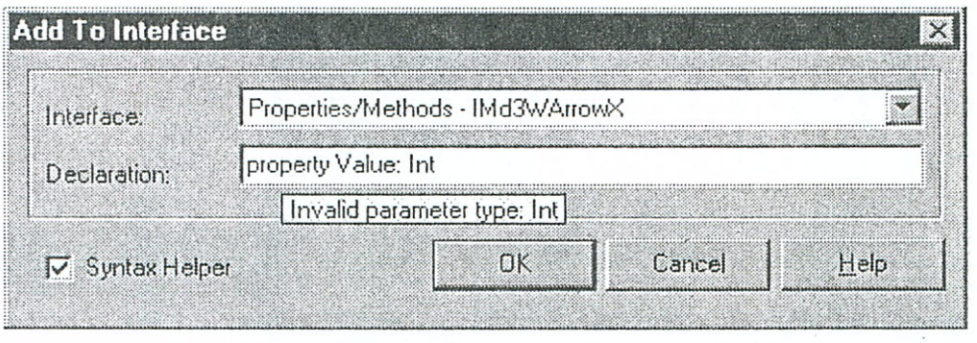
23.7. ábra *Az általunk készített ActiveX vezérlő, egy Delphi formon*

Mivel már megvan az importkönyvtár egység, ami tartalmazza a `TOleControl` alosztály deklarációját és regisztrációs információját, ezt a egységet egyszerűen hozzáadhatjuk egy új, vagy már létező csomaghoz. A telepítés végeztével a komponens megjelenik a komponenspalettán, az **ActiveX** lapon. Tegyünk egy példányt a komponensről egy formra, és máris szerkeszthetjük annak tulajdonságait, valamint az előugró menüben megtaláljuk a komponens Névjegy ablakát, ahogy az a 23.7. ábrán látható. Valójában ez az ActiveX vezérlő nem egészen pontos mása a VCL vezérlőnek. A VCL komponenseinek bonyolultabb tulajdonságai, mint például a TBrush vagy TPen, itt nem elérhetők.

Új tulajdonságok hozzáadása egy vezérlőhöz

Ahogy létrehozzuk az új ActiveX vezérlőt, hozzáadjuk az új tulajdonságokat, eseményeket és metódusokat meglepően tapasztalhatjuk, hogy ezt egyszerűbben megoldhatjuk, mint egy VCL komponens esetén. Ennek az oka az, hogy a Delphi az előbbihez speciális vizuális szerkesztési lehetőségeket is tartalmaz, míg az utóbbihoz nem.

Egyszerűen nyissuk meg az ActiveX implementációs részét tartalmazó Pascal egységet, és válasszuk az **Edit | Add To Interface** menüpontot, vagy használjuk a szerkesztő helyi menüjét. A Delphi erre egy Add to Interface nevű párbeszédablakot jelenít meg (23.8. ábra). A kombinált listából kiválaszthatjuk, hogy új tulajdonságot, metódust vagy esemény szeretnénk-e létrehozni. Az alábbi példában először az `IMd3WArrowX`, majd az `IMd3WArrowXEvents` interfészt.



23.8. ábra *Az Add to Interface párbeszédablak, amelyben a Syntax Helper éppen aktív*

A megjelenő szerkesztőmezőben kell meghatározni az új interfész elem deklarációját. Ha a Syntax Helper (szintaxis súgó) jelölőnégyzetet bekapcsoljuk, akkor tippeket kapunk, hogy az adott mezőt milyen formátummal kell kitölteni, és az esetleges hibákat ki is emeli a gép, ahogy azt a 23.8. ábrán láthatjuk.

Amikor új ActiveX interfész elemeket hozunk létre, ne felejtsük el, hogy ilyenkor csak OLE típusú adatokat használhatunk fel. Ebben az XArrow példában két új tulajdonságot hoztunk létre. Mivel az eredeti Delphi komponensek TBrush és a TPen tulajdonságai nem elérhetők, ezért ezeket újra láthatóvá kellett tenni. A következő programrész jó példa arra, hogy miket lehet az Add to Interface ablak szerkesztőmezőjébe beírni.

```
property FillColor: TColor;
property PenColor: Integer;
```

Megjegyzés

A TColor nem más, mint egy numerikus érték, az egész típusú változók egy fajtája. Ha megnézzük a típuskönyvtárat, akkor láthatjuk, hogy ott a USERDEFINED szerepel típusként, ami azt jelenti, hogy csak a Delphi tudja, hogy hogyan kell azt kezelni. Mindezek ellenére ennek a használata mellett döntöttem, és egy másik tulajdonságot egész típusúnak definiáltam, hogy láthassuk, ez a különbség hogyan jelentkezik egy másik fejlesztői környezetben.

Amit a párbeszédablakba beírunk, az automatikusan hozzáadódik a vezérlő típuskönyvtárához (.TLB fájl), az import fájlhoz, és az implementációs részhez:

```
type
  IMd3WArrowX = interface(IDispatch)
    function Get_FillColor: TColor; safecall;
    procedure Set_FillColor(Value: TColor); safecall;
    function Get_PenColor: Integer; safecall;
    procedure Set_PenColor(Value: Integer); safecall;
    ...
    property FillColor: TColor
      read Get_FillColor write Set_FillColor;
    property PenColor: Integer
      read Get_PenColor write Set_PenColor;
    ...
  DMd3WArrowX = dispinterface
    property FillColor: TColor dispid 8;
    property PenColor: Integer dispid 9;
    ...
  TMd3WArrowX = class(TActiveXControl, IMd3WArrowX)
    function Get_FillColor: TColor; safecall;
    procedure Set_FillColor(Value: TColor); safecall;
    function Get_PenColor: Integer; safecall;
    procedure Set_PenColor(Value: Integer); safecall;
    ...
```

Amint láthatjuk, ez nagyon nagy segítség a Delphi részéről, ezek kézi szerkesztése, és a három deklaráció szinkronizálása rengeteg munkát jelentene. Most már csak annyi dolgunk van, hogy az ActiveX vezérlő befejezett legyen, hogy a Get_x és Set_x metódus implementációs részét kitöltsük, hogy ezt itt is láthatjuk:

```
function TMd3WArrowX.Get_FillColor: TColor;
begin
  Result := FDelphiControl.Brush.Color;
end;

procedure TMd3WArrowX.Set_FillColor(Value: TColor);
begin
  FDelphiControl.Brush.Color := Value;
end;

function TMd3WArrowX.Get_PenColor: Integer;
begin
  Result := ColorToRGB (FDelphiControl.Pen.Color);
end;

procedure TMd3WArrowX.Set_PenColor(Value: Integer);
begin
  FDelphiControl.Pen.Color := Value;
end;
```

Ha ezt a vezérlőt most újra betöltjük a Delphibe, akkor már láthatóvá is válik annak két új tulajdonsága. Az első, a FillColor, TColor típusú tulajdonságként jelenik meg az Object Inspectorben, és a hozzá tartozó tulajdonságszerkesztő is használható. A Pen.Color egy egyszerű egész típusú tulajdonság, amiben elég nehéz a színt beállítani, így javaslom az RGB függvény használatát.

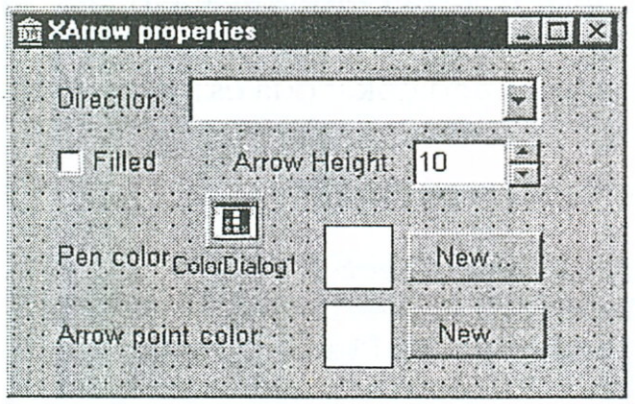
Más fejlesztői környezetben ez éppen fordítva működik, amely az egész típusú változót minden további nélkül felismeri, de a `TColor` típust nem. Ez a jelenség főleg olyan fejlesztőkörnyezetben jelentkezik, amelynek a Delphi Object Inspectorához hasonló ablaka van.

Tulajdonságlap hozzáadása a komponenshez

Más fejlesztői környezet nem tudja igazán jól használni komponensünket, mivel nem készítettünk hozzá tulajdonságlapot. Tulajdonságlap nélkül a komponens használhatatlan lenne, mert a felhasználó itt tudja a tulajdonságokat változtatni. Tulajdonságlap készítése azonban már nem annyira egyszerű feladat, mint új tulajdonságok létrehozása, mivel ez a gazda fejlesztői környezet integrált része. A saját vezérlőnk tulajdonságlapja az eredeti fejlesztői környezet tulajdonságlap ablakában fog megjelenni, amely kezeli a tulajdonságok megjelenítését, a billentyűk kezelését, OK, Cancel, Apply gombok megjelenítését.

Szerencsénkre a tulajdonságlap támogatása be van építve a Delphi 3-ba, ezért új lap hozzáadása meglehetősen egyszerű. Nyissuk meg az ActiveX projektet, a **File** menüben válasszuk ki a **New Item** menüpontot, és az ekkor megjelenő párbeszédablakból a **ActiveX** lapon válasszuk ki a Property Page-et. Amit most látunk, az nem sokban különbözik egy form-tól. Az alapértelmezés szerinti `TPropertyPage1` jelenik meg, ami a `TPropertyPage` leszármazottja, ami a `TCustomForm`-ból ered.

Ehhez a tulajdonságlaphoz ugyanúgy adhatunk komponenseket, mint egy hagyományos Delphi formhoz. Először egy kombinált listát adtunk a tulajdonságlaphoz, amiben a Direction tulajdonság lehetséges értékeit állítottuk be, majd egy jelölőnégyzetet a Filled tulajdonságnak, és egy szerkesztőmezőt egy léptetőgombbal, amin keresztül az ArrowHeight tulajdonságot fogjuk állítani, és két alakzatot a hozzá tartozó gombokkal, amikkel a színeket lehet változtatni. A 23.9. ábrán látható, hogyan néz ki a dolog szerkesztési időben.



23.9. ábra *Az XArrow ActiveX vezérlő tulajdonságlapja szerkesztési időben*

Már csak azt a kódrészletet kell elhelyeznünk a programban, ami biztosítja a színek szabályozására szolgáló két gomb helyes működését. Ezek az alakzatok mutatják, hogy milyen színű lesz ActiveX vezérlő. A ColorDialog komponens OnClick eseményének kezelése a szokásos módon történik:

```
procedure TPropertyPage1.ButtonPenClick(Sender:TObject);
begin
  with ColorDialog1 do
  begin
    Color := ShapePen.Brush.Color;
    if Execute then
    begin
      ShapePen.Brush.Color := Color;
      Modified; // enable Apply button!
    end;
  end;
end;
```

Fontos megjegyezni, hogy ha módosítás történik a formon, akkor a TPropertyPage Modified metódusát meg kell hívnunk, mert ekkor fog aktivizálódni az Apply gomb. Amikor a felhasználó valamely más vezérlőt használja a formon, akkor ez a metódus automatikusan lefut, de a két gomb estén magunknak kell ezt elvégezni.

Tipp

A tulajdonságlap címkéjének megválasztásakor ügyeljünk arra, hogy más fejlesztői környezetben ez a név a párbeszédablakon a tulajdonságlapnak megfelelő fül szövege lesz.

A következő lépés az, hogy a tulajdonságlap vezérlőit az ActiveX megfelelő tulajdonságaihoz rendeljük. A tulajdonságlapnak van erre két metódusa, az UpdateOleObject és az UpdatePropertyPage. Ahogy a nevük alapján is sejteni lehet, ezek a lap és az ActiveX között másolják az adatokat oda és vissza. Itt láthatjuk a példából a megfelelő kódrészt:

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
  { Update your controls from the OleObject }
  ComboDir.ItemIndex := OleObject.Direction;
  CheckFilled.Checked := OleObject.Filled;
  EditHeight.Text := IntToStr (OleObject.ArrowHeight);
  ShapePen.Brush.Color := OleObject.PenColor;
  ShapePoint.Brush.Color := OleObject.FillColor;
end;

procedure TPropertyPage1.UpdateObject;
begin
  { Update the OleObject from your controls }
  OleObject.Direction := ComboDir.ItemIndex;
  OleObject.Filled := CheckFilled.Checked;
  OleObject.ArrowHeight := UpDownHeight.Position;
  OleObject.PenColor := ColorToRGB(ShapePen.Brush.Color);
  OleObject.FillColor := ShapePoint.Brush.Color
end;
```

Az utolsó lépés az, hogy magát a tulajdonságlapot is összekapcsoljuk az ActiveX-szel. Amikor az ActiveX vezérlőt a Delphi létrehozza, automatikusan hozzáadja a DefinePropertyPage metódus deklarációját az implementációs egységhez. Ebben az eljárásban egyszerűen meghívjuk a DefinePropertyPage metódust minden olyan tulajdonságlapra, amit az ActiveX-hez akarunk adni. A metódus paramétere a tulajdonságlap GUID-je, amit a megfelelő egységben

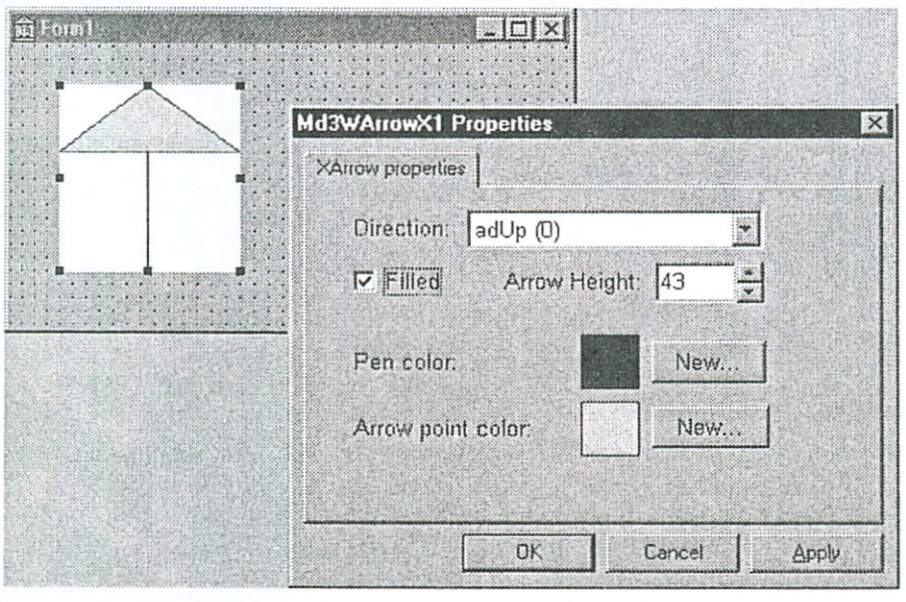
megtalálhatunk (természetesen a megfelelő egységre vonatkozó `uses` kifejezést tartalmaznia kell programunknak). Minderre a következő kódrészlet mutat példát:

```
procedure TMd3WArrowX.DefinePropertyPages(
  DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_PropertyPage1);
end;
```

Megjegyzés

Az ActiveX vezérlő és annak tulajdonságlapja közötti kapcsolat a GUID használatával jön létre. Ez azért lehetséges, mert a tulajdonságlap objektumot létre lehet hozni, habár a GUID és az osztálygyár a Windows regisztrációs adatbázisában van tárolva, amikor az adott ActiveX könyvtárat telepítjük. Hogy lássuk, hogyan is működik, nézzük meg a tulajdonságlap egységének inicializációs részét, ahol a `TActiveXPropertyPageFactory.Create` hívás található.

Most, miután befejeztük a tulajdonságlap szerkesztését, újrafordítottuk, és regisztráltuk az ActiveX könyvtárat, telepíthetjük az ActiveX vezérlőt a fejlesztői környezetben (például a Delphi-ben), és megnézhetjük, hogyan is működik. Erre mutat példát a 23.10. ábra (ha ezt az ActiveX vezérlőt előzőleg már telepítettük Delphi alá, akkor annak korábbi verzióját törölni kell, ehhez pedig elképzelhető, hogy újra kell indítanunk a Delphit).



23.10. ábra *Az XArrow ActiveX vezérlő, és annak tulajdonságlapja Delphi környezetben*

Hogyan tovább?

Ebben a fejezetben főleg az ActiveX vezérlőkkel foglalkoztunk. Megnéztük, hogyan kell őket használni, és miként teszi lehetővé Delphi 3 ezek egyszerű fejlesztését. Mindez igen fontos része a Delphinek, mivel alkalmasabbá teszi az alkalmazás magjának fejlesztésére, és lehetőséget nyújt a programozónak, hogy létrehozza a felhasználói felületet a komponenssel együtt, minden fejlesztői környezet számára.

A Delphi 3 az ActiveX vezérlők mellett az ActiveFormokat is támogatja. Ezek is ActiveX vezérlők, viszont másként kell fejleszteni és használni őket. Az ActiveFormok használatáról és fejlesztéséről, az Internet programozása mellett, a következő fejezetben lesz szó. Itt majd szó ejtünk azokról a megoldásokról, amelyek lehetővé teszik HTML oldalak megjelenítését és készítését, és az ezekkel kapcsolatos további témákban is elmélyedünk.

24. FEJEZET

Internetes programozás

E fejezet tartalma:

- HTML fájlok böngészése
- HTML fájlok készítése
- ActiveFormok létrehozása

Internetes programozás

Ebben a fejezetben a Delphi alatti Internet-közeli programfejlesztést szeretném röviden bemutatni. A Delphi 3-ban, és annak is Client/Server verziójában találunk néhány komponenst az internetes programok fejlesztésére, amelyek e témában széleskörű felhasználást biztosítanak. Ennek teljes áttekintése meghaladja egy könyv kereteit, de mindképpen szeretnék itt most bemutatni néhány érdekesebb lehetőséget.

Először HTML fájlokról lesz szó, megnézzük, hogyan kell írni egy egyszerű böngészőt, majd látunk példát HTML generátorokra is. A CGI vagy ISAPI technológiát felhasználó valós idejű HTML generálásra most nem térünk ki, csak ezen oldalak statikus készítésére. Attól függően, hogy milyen verziójú Delphit használunk, a fejlesztés történhet a Delphi saját komponenseivel, vagy külső cég által készítettekkel.

A fejezet végén szólunk még az ActiveForm-okról, és néhány korábbi példát fogunk ActiveForm-má alakítani. Ez az előző fejezet kiterjesztése, mivel az ActiveForm nem más, mint egy form, amit ActiveX-be csomagolunk, és egy ActiveX könyvtárba kerül.

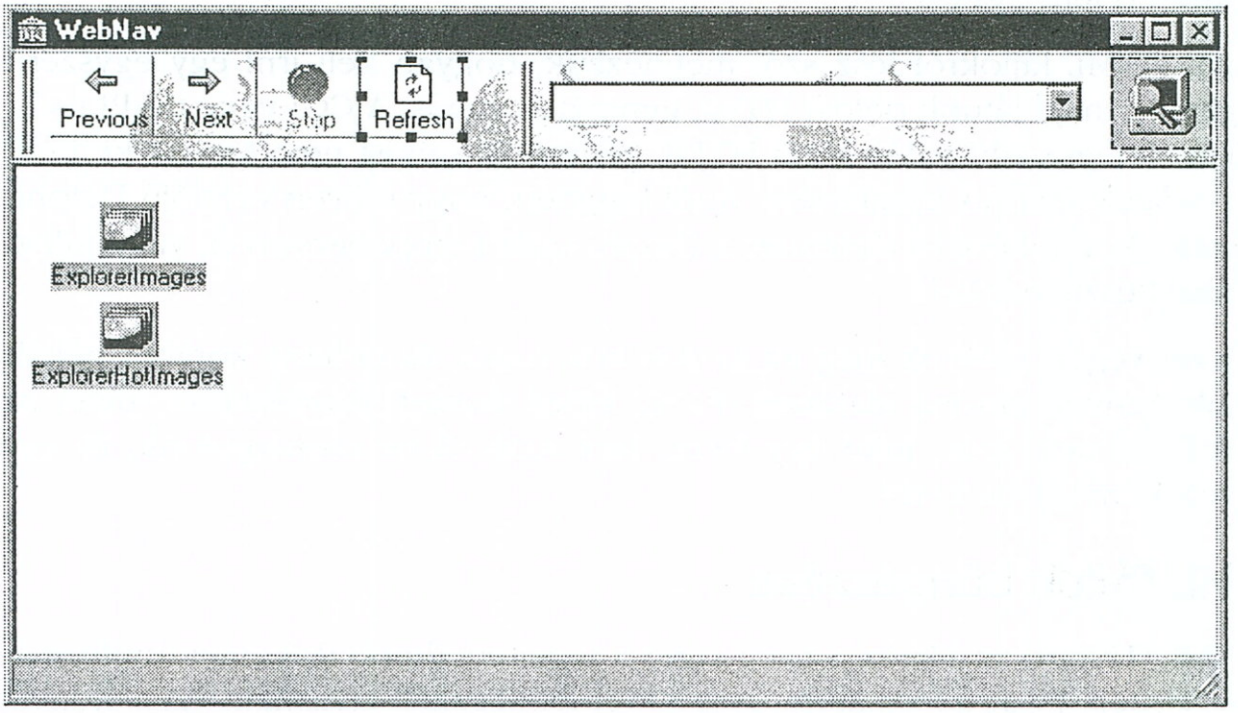
HTML fájlok böngészése

A HTML (Hypertext Markup Language) a legelterjedtebb dokumentumformátum a Weben. Ez az a formátum, amit a Web böngészők olvasnak, ilyen oldalak készítése és böngészése a fő feladat mindazok számára, akik evvel foglalkoznak. Mi ebben a részben először a már létező oldalak olvasásával foglalkozunk, majd ezt követően röviden szót ejtünk a HTML formátumról, mielőtt ezek szerkesztésébe kezdenénk.

A Delphi-ben ActiveX alapú Internet vezérlők egy csoportját találhatjuk meg a NetManage cég jóvoltából, amelyeket Internet Control Pack-nak hívunk. Ezek számos Internet protokollt felölelnek, mint például FTP, POP és UDP. A következő példában szeretnénk bemutatni, hogyan kell teljes mértékben működőképes Web böngészőt készíteni, a HTML vezérlő segítségével. Szükségünk lesz még egy eszköztárra, amelyen néhány funkciót lehet elérni. A HTML komponenst arra használni, hogy egy népszerű Web böngésző mását állítsunk elő, nem túl hasznos dolog. Ennek igazi előnye az, hogy különböző felhasználók igényeihez mérten testreszabhatjuk a böngészőt, illetve saját alkalmazásunkba be tudjuk építeni. Ez lehetőséget ad a felhasználónak, hogy lokális HTML oldalakat olvasson, vagy hálózati honlapokat nézegessen.

Ahogy az a WebNav példán keresztül is látszik, ez nem is olyan bonyolult feladat. Láthatunk középen egy HTTP vezérlőt, az ablak tetején egy speciális eszköztárat (Coolbar), és alul egy státuszsort. Ahogy az a 24.1. ábrán is látszik, a Coolbarnak van egy háttérképe, és további három komponens található rajta:

egy eszköztár, amin van négy gomb, ami a Web böngészők általános funkcióit látja el (Previous, Next, Stop és Refresh), egy kombinált lista, ami az URL (Uniform Resource Locator, az Internetes cím) megadását szolgálja, és tárolja a régebbi beírásokat, valamint egy Animate vezérlő. Ez a felület kísértetiesen hasonlít a Microsoft Internet Explorer-jére, de nem sokban különbözik a Netscape Navigator-tól sem.



24.1. ábra *A WebNav formja tervezési időben. Figyeljük meg az eszköztár és CoolBar vezérlőket*

Mivel a CoolBar és az eszköztár használatát a 11. fejezetben már megbeszéltük, erre most nem térek ki. Átugorjuk ezek tulajdonságainak tárgyalását, és a HTML vezérlő használatával foglalkozunk. Ennek alapvető eleme a RequestDoc metódus. Ezt a GotoPage eljárásban helyeztük el, mivel ennek meghívását sokszor fogjuk végezni:

```
procedure TFormWebNav.GotoPage (ReqURL: string);
begin
  // show the request on status bar and combo box
  ShowCurrent (ReqURL);
  // open the page
  HTML1.RequestDoc(ReqURL);
end;
```

A ShowCurrent metódus felel azért, hogy a felhasználói felület megfelelően legyen frissítve. A beírt URL-t a kombinált listához adja, ha az eddig nem

szerepelt benne, és a státuszsorban is megjeleníti azt, továbbá letiltja a Previous vagy a Next gombokat, attól függően, hogy a lista elején vagy végén vagyunk:

```
procedure TFormWebNav.ShowCurrent (ReqURL: string);
var
  Curr: Integer;
begin
  // add the item to the combo, if not present
  Curr := ComboURLs.Items.IndexOf(ReqURL);
  if Curr = -1 then
      Curr := ComboURLs.Items.Add(ReqURL);
  // select the items
  ComboURLs.Text := ReqURL;
  StatusBar1.Panels[0].Text := ReqURL;
  // enable toolbar buttons
  ToolBtnPrevious.Enabled := Curr > 0;
  ToolBtnNext.Enabled := Curr<ComboURLs.Items.Count-1;
end;
```

Miután a HTML vezérlő RequestDoc metódusát meghívtuk, a program kezel pár eseményt, mint például OnBeginRetrieval és OnEndRetrieval. Ebben a két eseményben egyszerűen az animációt indítjuk és állítjuk le, valamint engedélyezzük vagy letiltjuk a Stop gombot.

```
procedure TFormWebNav.HTML1BeginRetrieval(
    Sender:TObject);
begin
  ToolBtnStop.Enabled := True;
  Animate1.Active := True;
end;

procedure TFormWebNav.HTML1EndRetrieval(
    Sender: TObject);
begin
  ToolBtnStop.Enabled := False;
  Animate1.Active := False;
end;
```

Az ActiveX HTML vezérlő másik eseménye, amit még a program kezel, az OnDoRequestDoc esemény, amely akkor fut le, amikor a felhasználó a HTML dokumentumban kattint. Ebben ez esetben nekünk szükségünk van az új URL-re, amit a felhasználó kiválasztott, hogy azt a kombinált listába tudjuk helyezni:

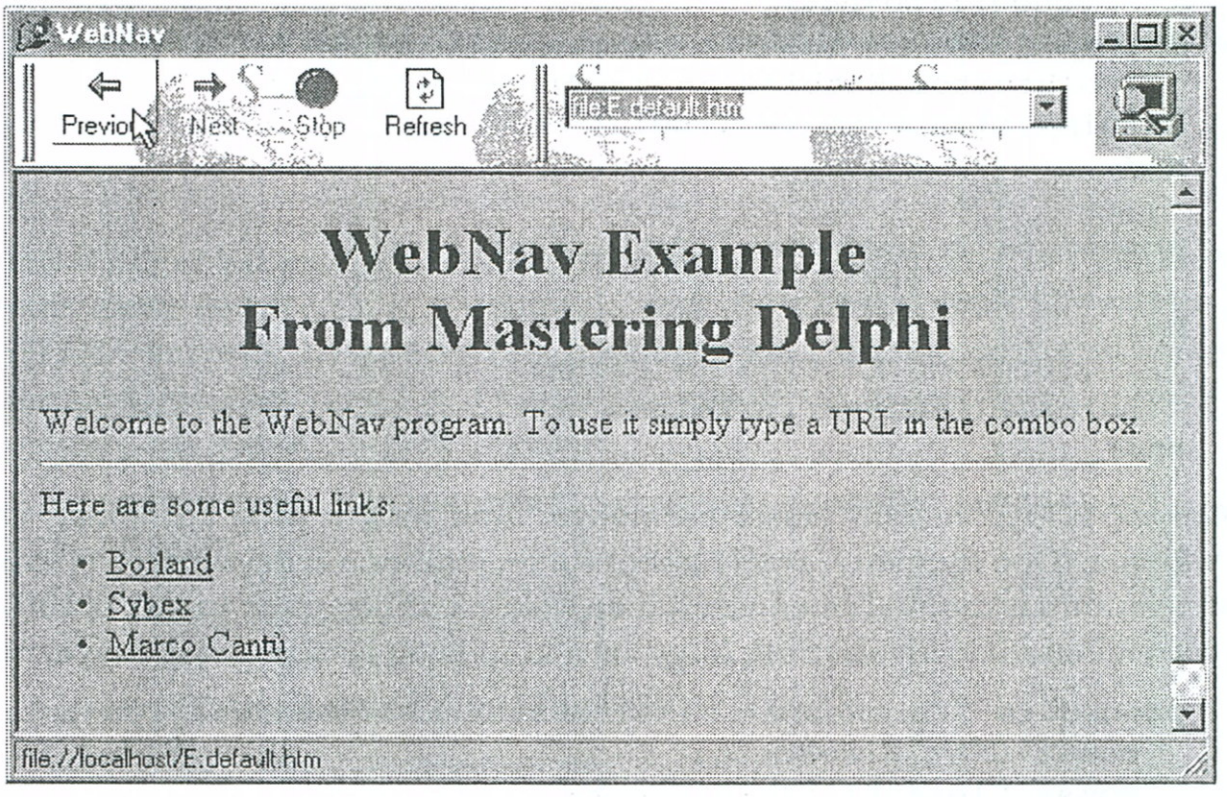
```
procedure TFormWebNav.HTML1DoRequestDoc(Sender: TObject;
  const URL: WideString;
  const Element: HTMLElement;
  const DocInput: DocInput;
  var EnableDefault: WordBool);
```

```
begin
  // show the current page
  ShowCurrent (URL);
end;
```

Ennek az eseménynek meglehetősen sok paramétere van, de nekünk ebből most csak az URL-t tartalmazó WideString típusú változóra van szükségünk. Most térjünk vissza a program elejére, és nézzük meg, hogyan is indul. Amikor a form létrejön a WebNav program betölti az aktuális könyvtárban lévő DEFAULT.HTM oldalt, ami a program kezelésével kapcsolatos információt tartalmaz:

```
procedure TFormWebNav.FormCreate(Sender: TObject);
var
  Drive: string;
begin
  Drive := ExtractFileDrive (ParamStr (0));
  GotoPage ('file:' + Drive + 'default.htm');
end;
```

Furcsa, de ahhoz, hogy az aktuális könyvtárból betöltsük egy oldalt, nem kell megadni a teljes könyvtárstruktúrát, viszont a meghajtót igen, különben a program csak a C meghajtóról működik. A fájl betöltését lehet látni a 24.2. ábrán.



24.2. ábra *A WebNav által megjelenített HTML fájl, az alkalmazás indulásakor*

Miután a nyitóoldal megjelenik, a kombinált listába be tudjuk írni az általunk megtekinteni kívánt webcímet, és ENTER-t nyomva a címet a HTML vezérlő megkapja, és megkezdi annak letöltését:

```
procedure TFormWebNav.ComboURLsKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if Key = VK_Return then
    GotoPage (ComboURLs.Text);
end;
```

A GotoPage eljárás akkor is lefut, amikor a kombinált lista OnClick eseménye meghívásra kerül, tehát a felhasználó a listából választ. Ezek után már csak az eszközsoron lévő gombok eseményeit kell kezelni, amelyre itt találjuk a példát:

```
procedure TFormWebNav.ToolBtnPreviousClick(
  Sender: TObject);
var
  Curr: Integer;
begin
  Curr := ComboURLs.Items.IndexOf(ComboURLs.Text);
  if Curr > 0 then
    GotoPage (ComboURLs.Items [Curr - 1]);
end;

procedure TFormWebNav.ToolBtnNextClick(Sender: TObject);
var
  Curr: Integer;
begin
  Curr := ComboURLs.Items.IndexOf(ComboURLs.Text);
  if Curr < ComboURLs.Items.Count - 1 then
    GotoPage (ComboURLs.Items [Curr + 1]);
end;

procedure TFormWebNav.ToolBtnStopClick(Sender: TObject);
begin
  HTML1.Cancel(Null);
  ToolBtnStop.Enabled := False;
end;

procedure TFormWebNav.ToolBtnRefreshClick(Sender:
TObject);
begin
  GotoPage (ComboURLs.Text);
end;
```

A fenti ToolBtnStopClick metódusban nem kell az animációt leállítani, mert ahogy a HTML komponens Cancel metódusát meghívjuk, az OnEndRetrieval esemény lefut, ami meg fogja állítani az animációt.

Viszonylag kevés munkával készítettünk Web böngészőt, igaz egy elég egyszerűt. Még egyszer szeretném kihangsúlyozni, hogy a példa célja nem egy minden funkciót ismerő böngésző készítése, hanem a HTML komponens bemutatása volt, és hogy megmutassuk, hogyan lehet ezt saját alkalmazásunkba beépíteni.

HTML fájlok készítése

A Weben, a kliens oldalon a HTML oldalak böngészése a fő feladat, a szerver oldalon pedig ezen oldalak készítése. Néha elég, ha egy adatbázis alapján készítünk el HTML oldalakat, amelyek nagyrészt megegyeznek egymással, máskor viszont dinamikusan a felhasználó igényei szerint kell honlapokat létrehozni. Először lássunk pár példát az első estre, aztán beszélek a dinamikus készítésről is.

A következőkben röviden áttekintjük a HTML fájl utasításainak egy igen szűk körét. E rész átugrásáról mindenki tudása alapján döntsön.

A HTML formátum

A HTML fájl nem más, mint egy ASCII fájl. A sima szöveg mellett viszont HTML tagokat (más néven címkéket) is találunk benne, amelyek meghatározzák a betűtípust, méretet, bekezdés beállításokat, képeket, más HTML oldalakhoz való kapcsolatokat, és egyéb jellemzőket.

A legtöbb tag páros, mint például a stílus, ahol az első tag jelzi, hogy annak hatásköre hol kezdődik, a párja pedig (ami egy / jel után megismétli az első tag első szavát) a hatáskör végét jelzi. Például a `<b>fontos</b>` részlet a fontos szót félkövér betűkkel jeleníti meg a böngésző, a `<title>Cím</title>` részlet pedig a HTML oldal címét állítja „Cím"-re.

Bizonyos tagok viszont önmagukban állnak (nincs hatókörük), mint például a `<p>`, ami új bekezdést kezd. Ez a címke azért is nagyon fontos, mert a HTML fájlban a szóközöknek és az újsor karaktereknek nincs jelentősége, éppen ezért `<p>` címkével tudjuk a kiírást új sorban folytatni.

A HTML dokumentum a `<html>` taggal kezdődik, és két fő részre osztható, amit a `<head>` és `<body>` taggal jelzünk. Ezek is páros tagok, és párjaik használata szükséges. A head (fejléc) részben általában a HTML fájl címét adjuk meg, ami a böngésző címsorában megjelenik, továbbá egyéb elemeket.

A body (törzs) részben van a fájl tényleges adattartalma, ami általában a dokumentum látható címével kezdődik. Megadhatunk fejléceket is,

A HTML formátum (folytatás)

különböző szinttel, amelyet <hX> taggal jelzünk, ahol X értéke 1-től 6-ig terjedhet. Ezek után jönnek a paragrafusok <p>, vagy a listák <pre> taggal jelezve.

A szövegbe gyakran kell olyan kapcsolatokat helyezni, amelyek más HTML oldalra, vagy ugyanazon oldal más részére mutatnak. Ezen részek tárgyalása nem ennek a könyvnek a feladata, ebben a témába rengeteg könyv létezik, de a Weben böngészve is találhatunk megfelelő anyagok, illetve HTML oldalak forrását tanulmányozva is sok ötletet kaphatunk.

Egy egyszerű HTML oldal generálása

HTML oldal készítésére az első példa a 22. fejezetben megismert DbToWord program lesz bizonyos módosításokkal, ami eredetileg adatbázis információkat küldött Microsoft Word-nek OLE-n keresztül. Most ugyanezt a funkciót úgy módosítjuk, hogy az adatbázisból HTML oldalt készítünk, és azt a Web böngészőnek adjuk át. Egyelőre egy egyszerűbb megoldással kezdjük, majd a következőkben ezt tökéletesítjük.

Ennek az új példának, amit DbToHTML-nek neveztem el, a formján találunk pár új komponenst, amelyek az előzőekben nem volt megtalálhatók. Az adatbázis-kezelő komponenseken kívül találunk három új gombot, amelyek egyéb speciális funkciót látnak el, egy jelölőnégyzetet, és egy Memo vezérlőt. A Memot arra használjuk, hogy az elkészült HTML forrását megjelenítsük benne. Valójában ennek további szerepe az, hogy a HTML forrását átmenetileg tárolja, mielőtt egy fájlba elmentjük.

Hogy bizonyos kódrészleteket később ne keljen megismételni, már most úgy írjuk meg őket, hogy azokat később módosítás nélkül fel tudjuk használni. Ilyen a HTML oldal elejét és végét elkészítő eljárás. Nagy előnyük, hogy nem úgy készültek, hogy a kizárólag Memo számára tárolják a HTML oldalt, hanem az eljárás paramétereként kap egy `TStrings` típusú objektumot, amihez hozzáadja a HTML forrását.

Az AddHeader metódus második paramétere egy string típusú változó, amely meghatározza, hogy a HTML címe mi legyen, ami a böngésző fejlécében és a HTML oldal tetején jelenik meg:

```
procedure TNavigator.AddHeader (Str: TStrings;
    Title: string);
begin
  Str.Add ('<HTML>');
  Str.Add ('<HEAD>');
```

```
    Str.Add ('<TITLE>' + Title + '</TITLE>');
    Str.Add ('</HEAD>');
    Str.Add ('<BODY>');
    Str.Add ('<H1><CENTER>' + Title + '</CENTER></H1>');
end;
```

A HTML vége még ennél is egyszerűbb. Nem tesz mást, mint megjelenít egy elválasztó vonalat (<HR> tag), megjeleníti a második sorban lévő szöveget, és lezárja a HTML dokumentumot:

```
procedure TNavigator.AddFooter (Str: TStrings);
begin
    Str.Add ('<HR>');
    Str.Add ('Generated by the program DbToHtml');
    Str.Add ('</BODY>');
    Str.Add ('</HTML>');
end;
```

És most lássuk az első gombhoz, a Print Line-hoz tartozó kódot. Amikor ez a gombot lenyomjuk, akkor a program törli a Memo tartalmát, meghívja az AddHeader eljárást, átadva neki az első rekordot, mint címet, majd minden rekordhoz megjelenít egy sort, és a lábléccel zárja a HTML-t:

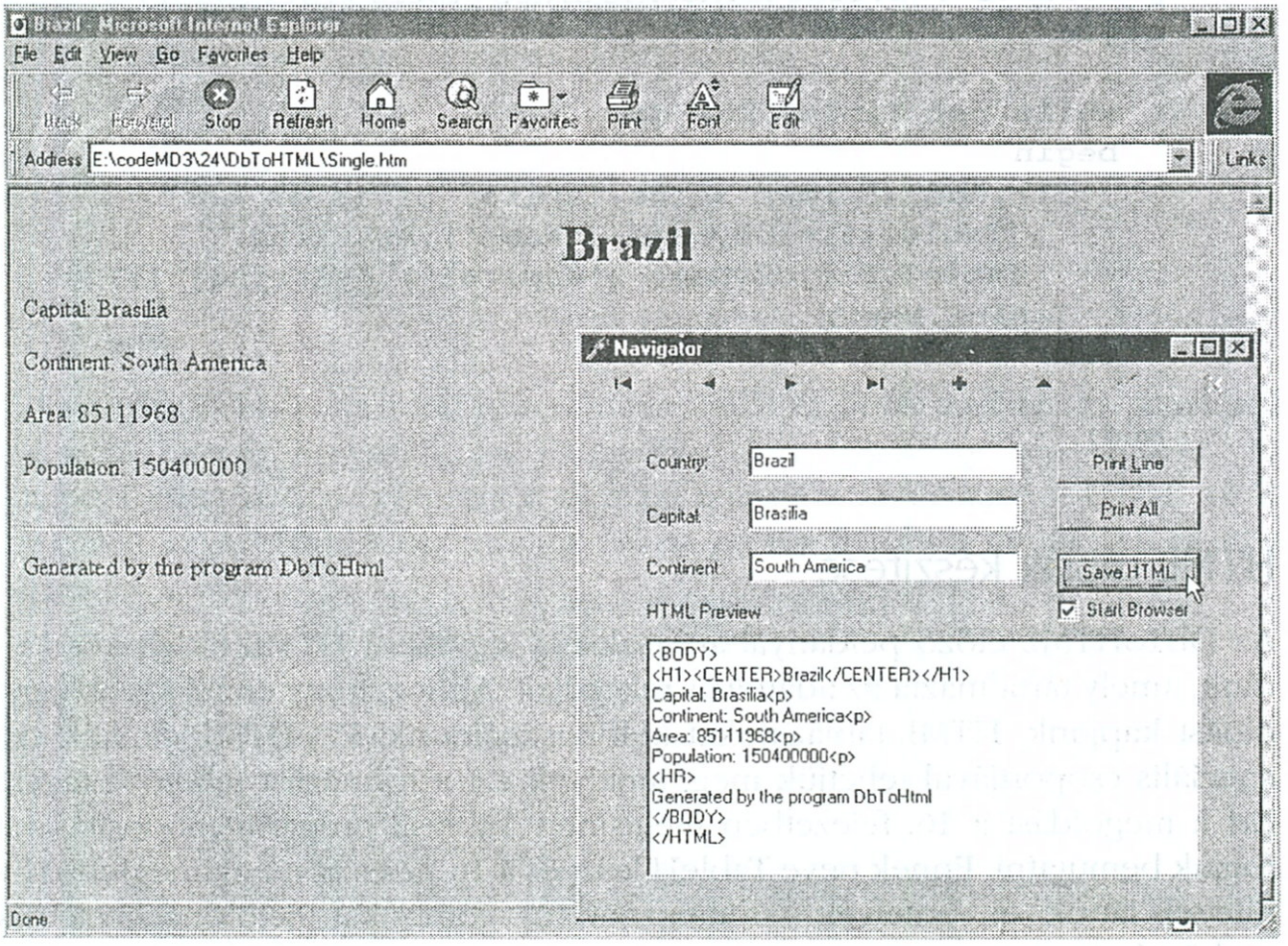
```
procedure TNavigator.BtnLineClick(Sender: TObject);
var
    I: Integer;
begin
    Memo1.Clear;
    AddHeader (Memo1.Lines, Table1.Fields[0].AsString);
    for I := 1 to Table1.FieldCount - 1 do
      Memo1.Lines.Add (Table1.Fields [I].FieldName+': '+
        Table1.Fields [I].AsString + '<p>');
    AddFooter (Memo1.Lines);
    BtnSave.Enabled := True;
end;
```

Amint láthatjuk, az előző eljárás amint elvégezte az oldal készítését, engedélyezi a Save gombot, ami ha megnyomják, megjelenít egy SaveDialog ablakot, és az ott megadott fájlba menti az elkészült HTML oldalt. A SaveDialog komponenst csak a HTM kiterjesztésű fájlok megjelenítésére állítottam be. A BtnSaveClick metódus utolsó ténykedése mielőtt befejeződne az, hogy ellenőrzi a Start Browser jelölőnégyzet állapotát, és ha az be van jelölve, akkor a HTML oldalt a rendszer alapértelmezett böngészőjében megjeleníti:

```
procedure TNavigator.BtnSaveClick(Sender: TObject);
begin
    if SaveDialog1.Execute then
    begin
      Memo1.Lines.SaveToFile (SaveDialog1.FileName);
```

```
    if CheckStart.Checked then
      ShellExecute (Handle, 'open',
        PChar (SaveDialog1.FileName), '', '',
          sw_ShowNormal);
  end;
end;
```

A ShellExecute használatával egyszerűen meg tudunk jeleníteni bármilyen dokumentumot. A Windows automatikusan megkeresi azt a programot, amelyik az adott kiterjesztéshez hozzá van rendelve, elindítja azt, átadva neki az elvégzendő műveletet (itt a megnyitást), és a fájlt. A 24.3. ábrán láthatjuk a programot működés közben.



24.3. ábra *A DbToHtml program automatikusan elkészíti a HTML forrását, és azonnal be is tölti a rendszer alapértelmezett böngészőjébe, ebben az esetben az Internet Explorer-be*

A program nem csak egy adott országra képes elvégezni a HTML generálását, hanem a teljes táblázatra, ahogy a DbToWord is tette, viszont ezt a Print All gomb metódusa végzi:

```
procedure TNavigator.BtnPrintClick(Sender: TObject);
begin
  Memo1.Clear;
  AddHeader (Memo1.Lines, 'Table: '+Table1.TableName);
  AddAllLines (Memo1.Lines);
  AddFooter (Memo1.Lines);
  BtnSave.Enabled := True;
end;
```

A program létrehozza a fejlécet, a láblécet, és a kettő között meghívja az AddAllLines metódust. Ez nem más, mint egy `while` ciklus, amely kiolvassa a teljes adatbázist, miután letiltotta a vezérlőket és tárolta a könyvjelzőt. Itt a teljes metódus helyett csak a ciklus látható:

```
procedure TNavigator.AddAllLines (Str: TStrings);
begin
  ...
  while not Table1.EOF do
  begin
    Str.Add (Format ('The capital of %s is %s<p>',
      [Table1.FieldByName ('Name').AsString,
      Table1.FieldByName ('Capital').AsString]));
    Table1.Next;
  end;
  ...
end;
```

HTML táblák készítése

Az DbToHTML előző példányában csak egy egyszerű HTML oldalt készítettünk, amely tartalmazta az adatbázis rekordjait. Ahhoz, hogy ennél szebb megoldást kapjunk, HTML táblázatokat kell használnunk. Ezt HTML címkék egy speciális csoportjával tehetjük meg, amelyek ezt a formázást lehetővé teszik. Ezt a megoldást a 16. fejezetben megismert Tables2 program módosításával fogjuk bemutatni. Ennek neve TableH lesz, és a 16.22-es ábrának megfelelő felülettel láttuk el, amelyek tartalmaznak új gombokat, jelölőnégyzetet és eszköztárat.

A HTML oldal készítéséhez szükséges kód hasonló az előző példában megismerthez. Viszont az előzőhöz képest bizonyos fejlesztéseket is alkalmaztunk, mivel itt nem találunk Memo komponenst, helyette egy `TStringList` objektumban tároljuk a HTML forrását. Ehhez az objektumhoz írtam egy új egységet, és úgy döntöttem, hogy ezt még tovább fejlesztem, és létrehozok egy új osztályt a `TStringList`-ből, amely már rendelkezik ezekkel az új képességekkel. Ennek az új osztálynak a struktúrája látható a következő:

```
type
  THtmlStrings = class (TStringList)
  public
    procedure AddHeader (Title: string);
    procedure AddFooter;
    procedure OutputTable (Data: TDataSet);
  private
    procedure AddTableContents (Data: TDataSet);
  end;
```

A privát metódus csak az OutputTable metódust segíti, és ezért más osztályokból nem hívható meg. Az AddFooter metóduson is változtattunk, hogy az adott program információja pontosabban jelenjen meg a HTML fájlban, de más tekintetben megegyezik az előző példában látottakkal:

```
procedure THtmlStrings.AddFooter;
begin
  Add ('<HR>');
  Add ('Generated by the program ' +
    ExtractFileName (ParamStr(0)));
  Add ('</BODY>');
  Add ('</HTML>');
end;
```

A ParamStr(0) hívás az adott program útvonalát és nevét adja vissza, amelyből mi a fájl nevét kinyerjük az ExtractFileName függvény segítségével. Az OutputTable metódus elkészíti a táblázat fejlécét a fejléc azonosítóival, aztán meghívja az AddTableContents metódust, amely feltölti a táblázatot, majd a végén zárja azt:

```
procedure THtmlStrings.OutputTable (Data: TDataSet);
var
  I: Integer;
begin
  // start table with borders
  Add('<table border>');
  // new row, with the table headers (tag <th>)
  Add('<tr>');
  for I := 0 to Data.FieldCount - 1 do
    if Data.Fields[I].Visible then
      Add('<th>' + Data.Fields[I].FieldName + '</th>');
  Add('</tr>');
  // new row for each record, with the proper fields
  AddTableContents (Data);
  // done
  Add('</table>');
end;
```

Megjegyzés

HTML táblázatok készítése nem igényel bonyolult beállításokat, habár sok olyan opció létezik, amelyekről most nem teszünk említést. A legfontosabb tag a `<table>` ami egy új táblázatot nyit, az alapértelmezett szegélybeállításokkal. A `<tr>` tag új sort, a `<th>` illetve `<td>` fejléc illetve adatoszlopot kezd. Az egy sorban lévő oszlopok száma nincs megszabva, attól függ mennyit nyitunk belőlük, így ugyanabban a táblázatban lehetnek különböző számú oszlopot tartalmazó sorok. Ebben a példában minden rekordnak egy külön sort nyitunk.

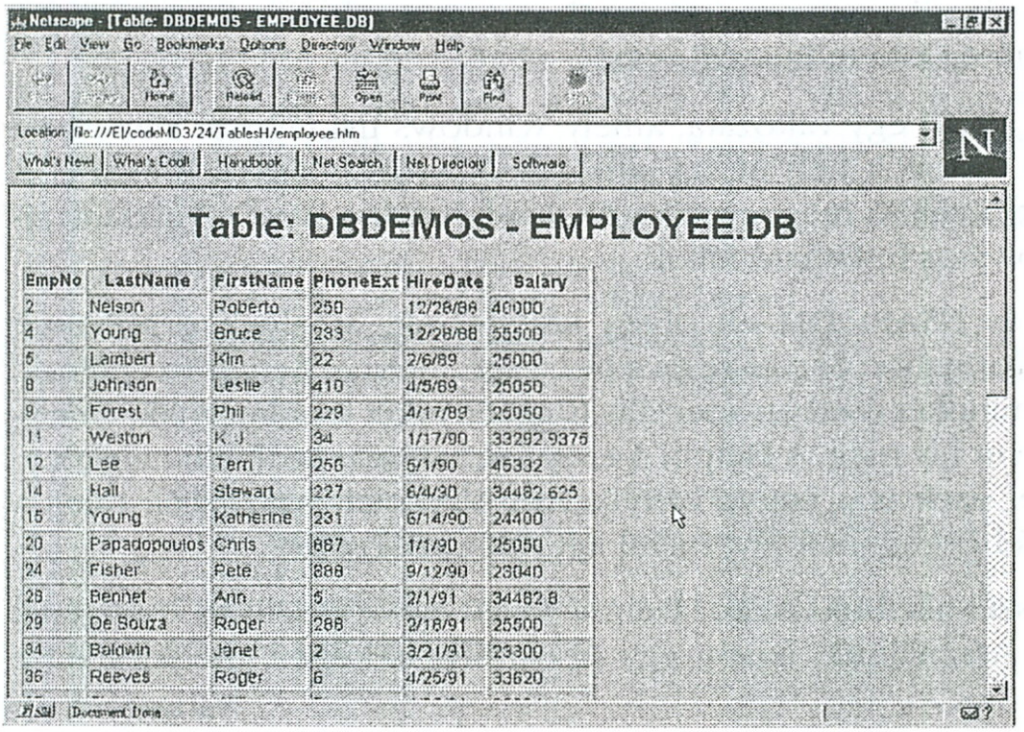
Az AddTableContents metódus szintén egy `while` cikluson alapszik, ami a táblázatot olvassa. Itt látható ennek központi része:

```
procedure THtmlStrings.AddTableContents (Data:TDataSet);
begin
  ...
  while not Data.EOF do
  begin
    Add('<tr>'); // new row, with table data (tag <td>)
    for I := 0 to Data.FieldCount - 1 do
      if Data.Fields[I].Visible then
        Add('<td>' +
            Data.Fields[I].DisplayText + '</td>');
    Add('</tr>');
    Data.Next;
  end;
  ...
end;
```

Amint ez elkészült, a egységet a HtmlStr.pas állományba mentjük. Ezek után a program fejlesztése meglehetősen egyszerű. Amit még meg kell írnunk, az az új SpeedButton OnClick eseményének kezelése. Ez a metódus először megkérdezi a felhasználótól az új HTML fájl nevét, és erre javaslatot is tesz a táblázat neve alapján (persze HTM kiterjesztéssel). Ezek után létrehozza a `THtmlStrings` objektumot, és meghívja annak három metódusát, menti az elkészült HTML forrást, és megsemmisíti az objektumot (ezt is egy védett blokkban, arra az esetre, ha valami hiba történne). Az új osztálynak köszönhetően az így elkészült kódrész meglehetősen egyszerű, és könnyen emészthető:

```
procedure TMainForm.SpeedButton2Click(Sender: TObject);
var
  Str: THtmlStrings;
begin
  SaveDialog1.FileName := ChangeFileExt (
    Table1.TableName, '.htm');
  if SaveDialog1.Execute then
  begin
    Str := THtmlStrings.Create;
    try
      Str.AddHeader (Caption);
      Str.OutputTable (Table1);
      Str.AddFooter;
      Str.SaveToFile (SaveDialog1.Filename);
      if CheckBox1.Checked then
        ShellExecute (Handle, 'open',
          PChar (SaveDialog1.FileName),
          '', '', sw_ShowNormal);
    finally
      Str.Free;
    end;
  end;
end;
```

Ha a jelölőnégyzet be van jelölve, akkor a program a HTML oldal elkészítse után a rendszer alapértelmezett böngészőjében automatikusan meg is jeleníti azt, ahogy ezt a 24.4. ábrán láthatjuk.



EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary
2	Nelson	Roberto	250	12/28/88	40000
4	Young	Bruce	233	12/28/88	55500
5	Lambert	Kim	22	2/6/89	26000
8	Johnson	Leslie	410	4/5/89	25050
9	Forest	Phil	229	4/17/89	25050
11	Weston	K J	34	1/17/90	33292.9375
12	Lee	Terri	256	5/1/90	45332
14	Hall	Stewart	227	6/4/90	34482.625
15	Young	Katherine	231	6/14/90	24400
20	Papadopoulos	Chris	887	1/1/90	25050
24	Fisher	Pete	888	9/12/90	28040
28	Bennet	Ann	5	2/1/91	34482.8
29	De Souza	Roger	288	2/18/91	25500
34	Baldwin	Janet	2	3/21/91	23300
36	Reeves	Roger	6	4/25/91	33620

24.4. ábra *A TableH által generált HTML oldal, a Netscape Navigator-ban*

HTML oldalak kívánság szerinti készítése

Az eddigi példákban, olyan megoldásokkal foglalkoztunk, ahol a HTML oldalakat azelőtt készítettük el, mielőtt feltöltöttük volna a szerverre. Ez nagyban különbözik attól, amikor lehetővé tesszük, hogy a szerver fogadja a felhasználó kívánságát, és annak megfelelően, „interaktívan" hozza létre a HTML oldalt, és azt a felhasználóhoz továbbítsa.

Alapvetően ennek megoldására két technika létezik: az egyik a Common Gateway Interface (CGI) használata annak valamely formájában, vagy egy DLL készítése, amely a szerverre kapcsolódik. Mind a két esetben ahelyett, hogy a keresett fájl útvonalát és nevét adjuk át a szervernek, a program (vagy DLL) nevét adjuk meg, néhány egyéb paraméter kíséretében.

Delphi lehetővé teszi, hogy ilyen szerver-alkalmazásokat készítsünk, amelyek majd Windows alapú szervereken futtathatók. A Delphi 3 Client/Server változata ehhez külön segítséget is nyújt. A megfelelő varázslót használva elkészíthetjük az alkalmazás vázát, legyen az CGI, WinCGI vagy DLL. Ha ez nem található az általunk használt Delphi verzióban, akkor külső cég által készített termékre kell hagyatkoznunk. A következőkben az előbbiek mindegyikére mutatunk példát, de ezek csupán rövid ízelitőt a témáról.

- A CGI, vagy Common Gateway Interface az Interneten rendkívül elterjedt technika a kliens részéről bejövő adatok kezelésére, és ezt a legtöbb Web szerver támogatja is. Alapjában véve a szerver a CGI programmal kommunikál környezeti változókon keresztül, és a felhasználói inputot paraméterekkel oldja meg. Ez a megoldás a UNIX bemeneti/kimeneti modelljén alapul, ami nem túl hatékony. CGI alkalmazásokat önálló konzolalkalmazásként fejleszthetünk.
- A WinCGI a CGI egy változata, amely Windows INI fájlokat alkalmaz a környezeti változók helyett, az információ továbbadására. Ennek ellenére a megközelítés megegyezik a CGI-vel, habár a WinCGI alkalmazások Windows-os futtatható fájlok.
- Az ISAPI és NSAPI DLL-ek olyan könyvtárak, amelyek a Microsoft Internet Information Server-re és a Netscape Internet Server-re épülnek. A DLL-ek mindkét esetben speciális függvényeket tartalmaznak, amelyeket a szerver hív meg. A Delphi 3 ezek fejlesztését rendkívül megkönnyíti, mivel ezek is a hagyományos API hívásokat használják (amit a Borland WebBridge-nek hív, mivel két API-t köt össze).

Mindhárom módszer alkalmazásakor Delphi 3 alatt egy speciálisan konfigurált `TDataModule`-t használunk, amely egy új osztályt is képez, melynek neve WebModule.

ActiveFormok létrehozása

Az 1. fejezetben egy pillantást vethettünk arra, hogy milyen egyszerű Delphi 3 alatt ActiveX alapú formokat létrehozni, és azt az Interneten telepíteni. Természetesen az így kapott fájlt olyan böngészővel lehet csak futtatni, amely támogatja az ActiveX technológiát. Ebben a részben szeretnénk átismételni az ActiveFormok készítését, kitérve bizonyos technikai részletekre, és ezt néhány példán keresztül bemutatni.

Az ActiveFormok fejlesztése változatlanul a megfelelő varázsló segítségével történik. Természetesen megtehetnénk mindezt varázsló nélkül is, jelentős többletmunkával. A varázsló hátránya főleg akkor jelentkezik, amikor egy már meglévő programot akarunk ActiveForm alapúvá alakítani, mert akkor annak teljes kódját az új ActiveForm-ba be kell másolni.

A példánkban létrehoztunk egy új ActiveLibrary-t, majd egy új ActiveForm-ot. Ennek a példának a neve XForm1 lesz, amiben a Delphi által generált kódból indulunk ki, majd adunk hozzá néhány saját tulajdonságot is. Ez első szembetűnő különbség egy sima Delphi form-hoz képest a deklarációban van, ahol láthatjuk, hogy az új ActiveForm a `TActiveForm`-ból származik:

```
type
  TAXForm1 = class(TActiveForm, IAXForm1)
```

Ahogy máskor is tapasztaltuk, az IAXForm interfész a típus könyvtárban deklarálódik, amely Pascal fájljának mi most a XF1Lib.pas-t adtuk. Ebből láthatunk itt egy kis részletet:

```
type
  IAXForm1 = interface(IDispatch)
    ['{51661AA1-9468-11D0-98D0-444553540000}']
    // Get and Set methods for TForm properties
    function Get_Caption: WideString; safecall;
    procedure Set_Caption(
        const Value: WideString); safecall;
    ...
    // TForm methods redeclared
    procedure Close; safecall;
    function CloseQuery: WordBool; safecall;
    ...
    // TForm properties
    property Caption: WideString
      read Get_Caption write Set_Caption;
```

Természetesen a varázsló létrehozza a `dispinterface`-t a form számára, és egyet az események számára:

```
type
  DAXForm1 = dispinterface...
  IAXForm1Events = dispinterface...
```

A TAXForm1 osztály számára generált kódban megtalálhatjuk a lekérdező és beállító metódusok implementációit, amelyek egyes tulajdonságok értékeit kérdezik le, vagy állítják, és ugyanitt vannak a form eseményeinek implementációi. Erre láthatunk itt egy rövid példát:

```
private
    FEvents: IAXForm1Events;
    procedure ActivateEvent(Sender: TObject);
    procedure ClickEvent(Sender: TObject);
  protected
    procedure EventSinkChanged(
      const EventSink: IUnknown); override;
    procedure Initialize; override;
    function CloseQuery: WordBool; safecall;
    function Get_Caption: WideString; safecall;
    procedure Close; safecall;
    procedure Set_Caption(
      const Value: WideString); safecall;
```

Figyeljük meg, hogy a form-nak van egy privát és egy védett része, továbbá az interfésznek is vannak tulajdonság definíciói, amelyek publikusként vannak megadva. Most először lássuk a tulajdonságok implementációit:

```
function TAXForm1.Get_Caption: WideString;
begin
  Result := WideString(Caption);
end;

procedure TAXForm1.Set_Caption(const Value: WideString);
begin
  Caption := TCaption(Value);
end;
```

Most pedig nézzük az eseményekét. A TForm eseményei szintén belső metódusként lettek deklarálva:

```
procedure TAXForm1.Initialize;
begin
  OnActivate := ActivateEvent;
  OnClick := ClickEvent;
  OnCreate := CreateEvent;
  ...
end;
```

Minden esemény egy külső ActiveX eseményt hív meg, ahogy azt a két következő példa is szemlélteti:

```
procedure TAXForm1.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;
```

```
procedure TAXForm1.ClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnClick;
end;
```

Éppen e hívások miatt, az eseményeket nem szabad a form eseményeiből kezelni, hanem ki kell egészíteni a leíróját, vagy egyszerűen felülírni a TForm metódusait, amelyek a végén meghívják az eseményeket (valójában ugyanezt az eljárást használjuk Delphi komponens készítésekor). Ne felejtsük el, hogy az ActiveForm interfész tulajdonságai fejlesztők számára vannak fenntartva, nem arra, hogy azt felhasználók használják a Weben. Mindez csak a form eseményeire vonatkozik, a komponensekére nem. A komponensek eseményeit ugyanúgy kezelhetjük, mint ahogy azt egyébként is tennénk.

Miután e belső részleteket megtárgyaltuk, az így létrejött vázat felhasználva írjunk egy kis példát, hogy a fenti problémát lássuk. Egyelőre csak az OnPaint eseményt szeretném kezelni. Az Object Inspector-ban az OnPaint eseményen duplán kattintva hozzuk létre az új leírót, majd a következő sorokkal egészítsük ki:

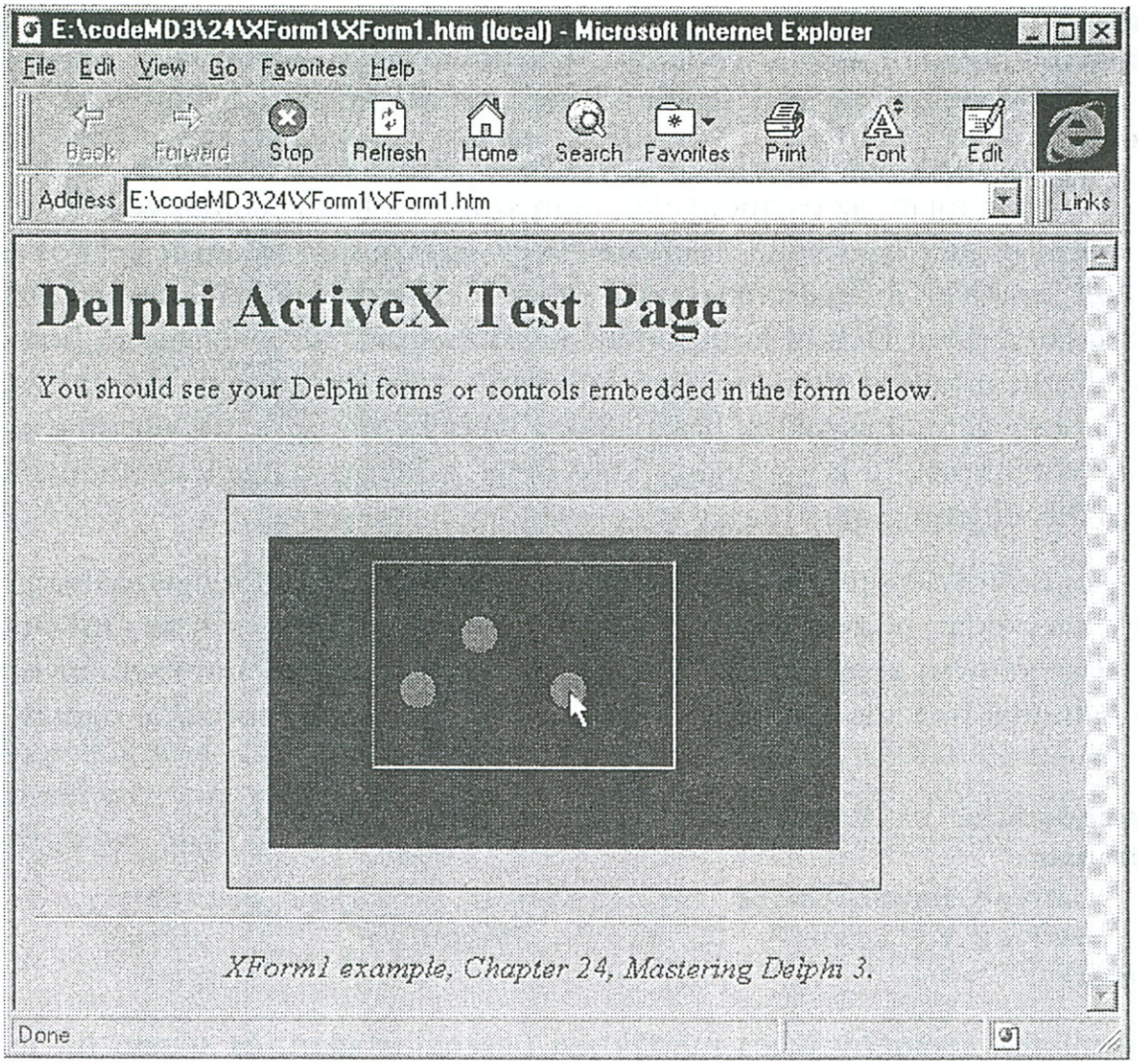
```
procedure TAXForm1.FormPaint(Sender: TObject);
begin
  Canvas.Brush.Color := clYellow;
  Canvas.Ellipse(0, 0, Width, Height);
end;
```

Ez a FormPaint esemény teljes mértékben használhatatlan, és sosem fut le. Ahhoz, hogy a form hátterére bármit is rajzoljunk, az ActiveForm Varázsló által létrehozott megfelelő leírót kell megváltoztatni:

```
procedure TAXForm1.PaintEvent(Sender: TObject);
begin
  Canvas.Brush.Color := clBlue;
  Canvas.Rectangle (20, 20,
    ClientWidth - 20, ClientHeight - 20);
  if FEvents <> nil then FEvents.OnPaint;
end;
```

Természetesen a formra helyezhetünk egy panelt, vagy más komponenst, és kezelhetjük annak az eseményét is. Ebben az esetben egy PaintBox-ot helyeztünk a formra egy Bevel komponenssel a háttérben, hogy a PaintBox határvonalát láthatóvá tegyük. Ezek után ennek a területnek a kezelése már jóval egyszerűbbé válik, ahogy azt a példa is mutatja (24.5. ábra):

```
procedure TAXForm1.PaintBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  PaintBox1.Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```



24.5. ábra *Az XForm1 ActiveForm az Internet Explorerben*

Figyelem!

Delphi alkalmazások írása legtöbbször egy interaktív folyamat. Először egy egyszerűbb verzióval indulunk, majd folyamatosan bővítjük újabb tulajdonságokkal, és közben ellenőrizzük működését. ActiveForm esetén viszont már jelentkezhetnek nehézségek. Az Internet Explorer az ActiveX vezérlőket a Windows Occache nevű mappájában tárolja, és annak ellenére, hogy egy OCX fájlt újrafordítunk, lehet, hogy a régi verziót fogja használni. Legcélravezetőbb megoldás az, ha az új példány futtatása előtt töröljük e könyvtár tartalmát.

ActiveX form szerepe egy weblapon

Mielőtt tovább haladnánk, fontos megemlíteni az ActiveX formok szerepét egy Web oldalon. Alapjában véve egy ilyen form Web oldalra helyezése nem jelent mást, mint hogy a felhasználónak lehetővé tesszük, hogy letöltsön és futtasson egy Windows-os programot. Ezen kívül nem sok minden történik. A program letöltődik, aztán végrehajtódik (ez az egyik oka, amiért az ActiveX technológia sok biztonsági kérdést vet fel, amire itt nem szeretnénk kitérni).

A következő példa ezt szeretné bemutatni. Létrehoztam egy ActiveFormot, tettem rá egy gombot és egy szövegmezőt, majd a gomb OnClick eseményére ezt a kódot írtam:

```
procedure TXUser.Button1Click(Sender: TObject);
var
  UserName: string;
  Size : Integer;
begin
  Size := 128;
  SetLength (UserName, Size);
  GetUserName (PChar(UserName), Size);
  Label1.Caption := UserName;
end;
```

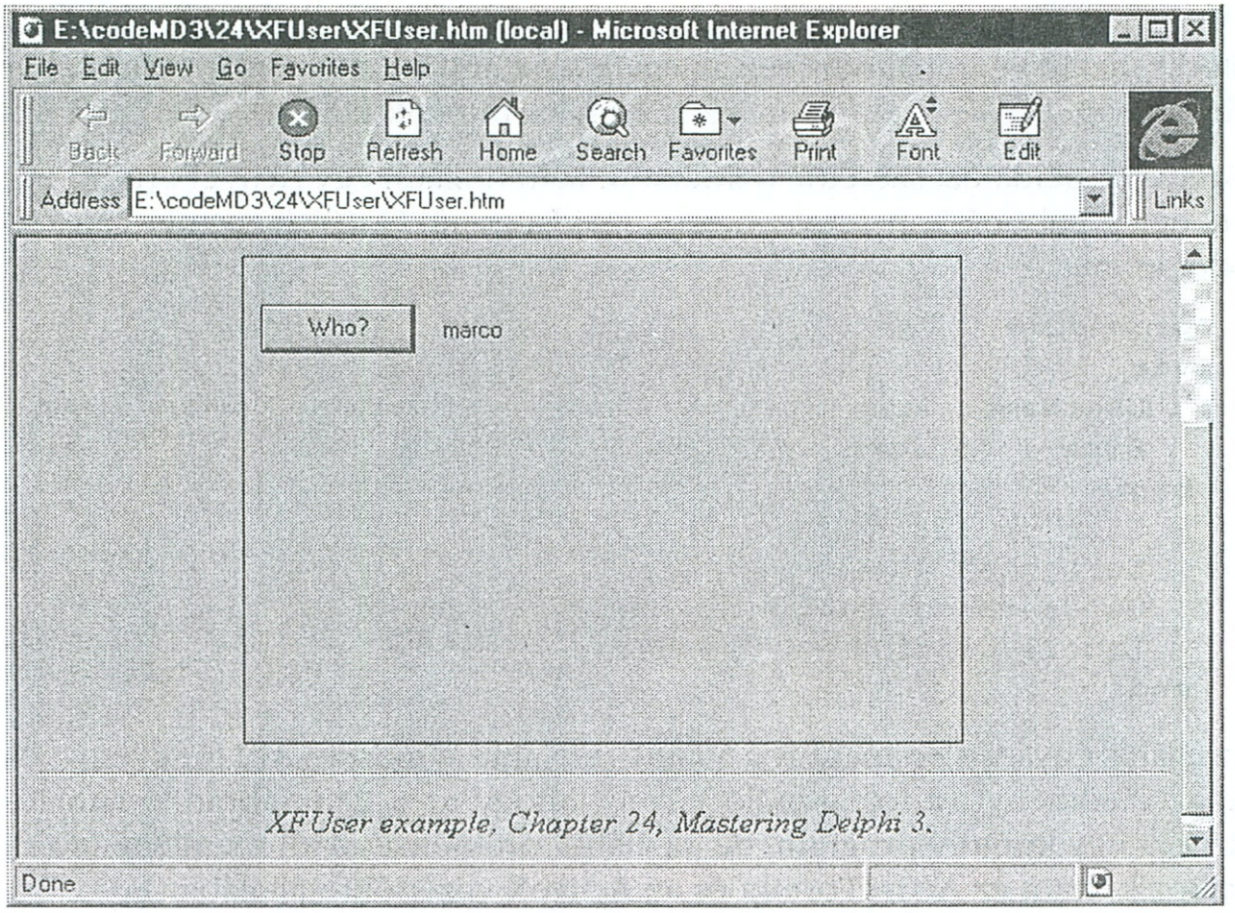
Ez az eljárás egyszerűen meghívja a GetUserName Windows API függvényt, és kiírja az eredményt (a felhasználó nevét), ahogy az a 24.6. ábrán is látható. Igaz, nem egy komoly program, de az alábbi problémákat jól szemlélteti (ami vonatkozik, mint az ActiveFormra és az ActiveX vezérlőkre általában):

- Egy ActiveX vezérlőből vagy formból bármely Windows API függvényt meg lehet hívni, ami azt jelenti, hogy a felhasználónak, aki az adott lapot nézi, Windows alapú gépen kell dolgoznia, vagy legalábbis Windows API kompatíbilis könyvtárakkal kell rendelkeznie.
- Egy ActiveX elérheti a rendszer információt, például a felhasználó nevét, vagy a könyvtárstruktúrát.
- Ezért van az, hogy mielőtt a böngésző egy ActiveX-et letölt, ellenőrzi annak megfelelő hitelességét, és aláírását. Ezek ugyan azonosítják az ActiveX szerzőjét, viszont nem biztosítják, hogy a vezérlő biztonságos.

Még folytathatnánk a felsorolást, de a lényeget már most is láthatjuk. Az ActiveX vezérlők és az ActiveFormok nagyszerű eszközök, főleg ha Intraneten használják őket. Interneten viszont egy ActiveX kontrol fogadtatása már vegyes lehet.

Az XFUser példában egy másik technikát szeretnénk bemutatni, amivel a **Web Deploy** (elhelyezés a hálózaton) pont használatakor találkozhatunk (a Delphi Client/Server változatánál). Itt lehetőségünk van CAB fájlok használatára,

amely olyan tömörített fájl, amiben benne vannak az OCX-ek és egyéb külső fájlok. Ezt könnyebb a felhasználó felé továbbítani, és mivel tömörített, gyorsabban tölthető le a Webről.



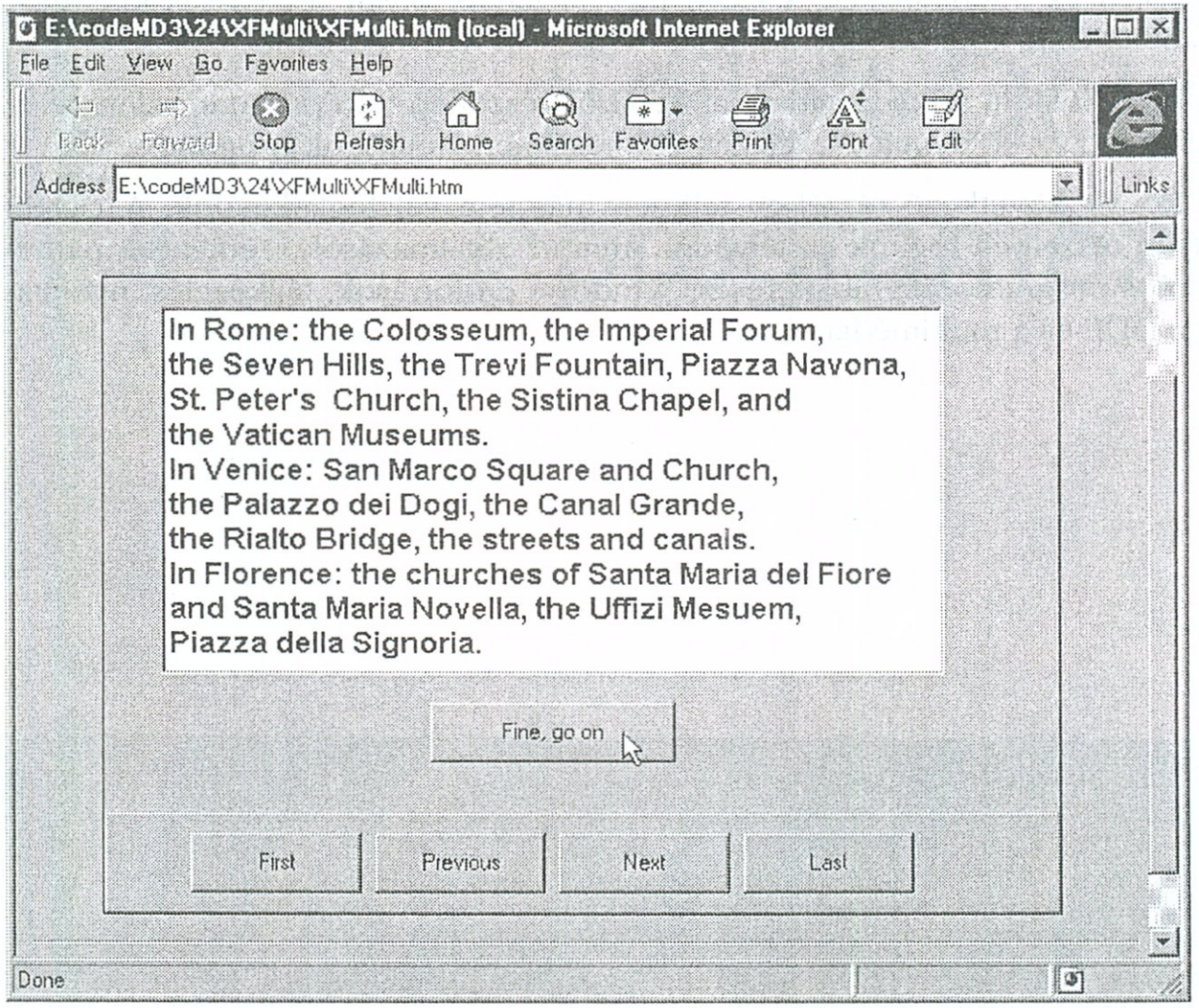
24.6. ábra *Ahogy azt a XFUser példa is mutatja, az ActiveForm a gépen bármilyen információt képes elérni*

Többlapos ActiveFormok

Utolsó példánkban egy régebbi programot alakítunk át ActiveFormmá. Amint azt már megtárgyaltuk, komplex programok készítésének, ahol több formot akarunk használni, három lehetősége van: MDI, több modális vagy nem modális form, vagy többlapos form alkalmazása. Ezek kövül ActiveFormok készítéséhez a legutolsó megoldás a leginkább célravezető.

Az XFMulti elkészítésekor egyszerűen a 14. fejezetben megismert NoteOnly példa kódjának nagy részét átmásoltuk az új példába. Ezt legegyszerűbben és talán legkönnyebben úgy tehetjük meg, hogy kijelöljük az eredeti példa komponenseit, majd abból készítünk egy komponens sablont, ami elérhetővé teszi a komponensekkel együtt annak tulajdonságait és eseményeit, és azt az új ActiveForm példába másoljuk.

Az elkészült programot láthatjuk a 24.7. ábrán, ahogy az a böngészőben fut. Ez ugyan egy egyszerű példa, viszont a PageControl komponens használatát nagyon jól bemutatja, amikor azt egy ActiveForm alkalmazásban kívánjuk alkalmazni.



24.7. ábra *A kész XFMulti példaprogram: a Web böngészőben futó program szinte teljesen úgy néz ki, mint a 14. fejezetben megismert NoteBook program*

Hogyan tovább?

Ebben a rövid fejezetben bemutattuk, hogyan kell Internettel kapcsolatos programokat készíteni. Ha ennél bővebb információra van szükségünk, találhatunk olyan könyveket, amelyek kimondottan ezzel a témával foglalkoznak. Itt a cél csak egy áttekintés volt a HTML és ActiveForm alapú alkalmazások fejlesztésére Delphi alatt.

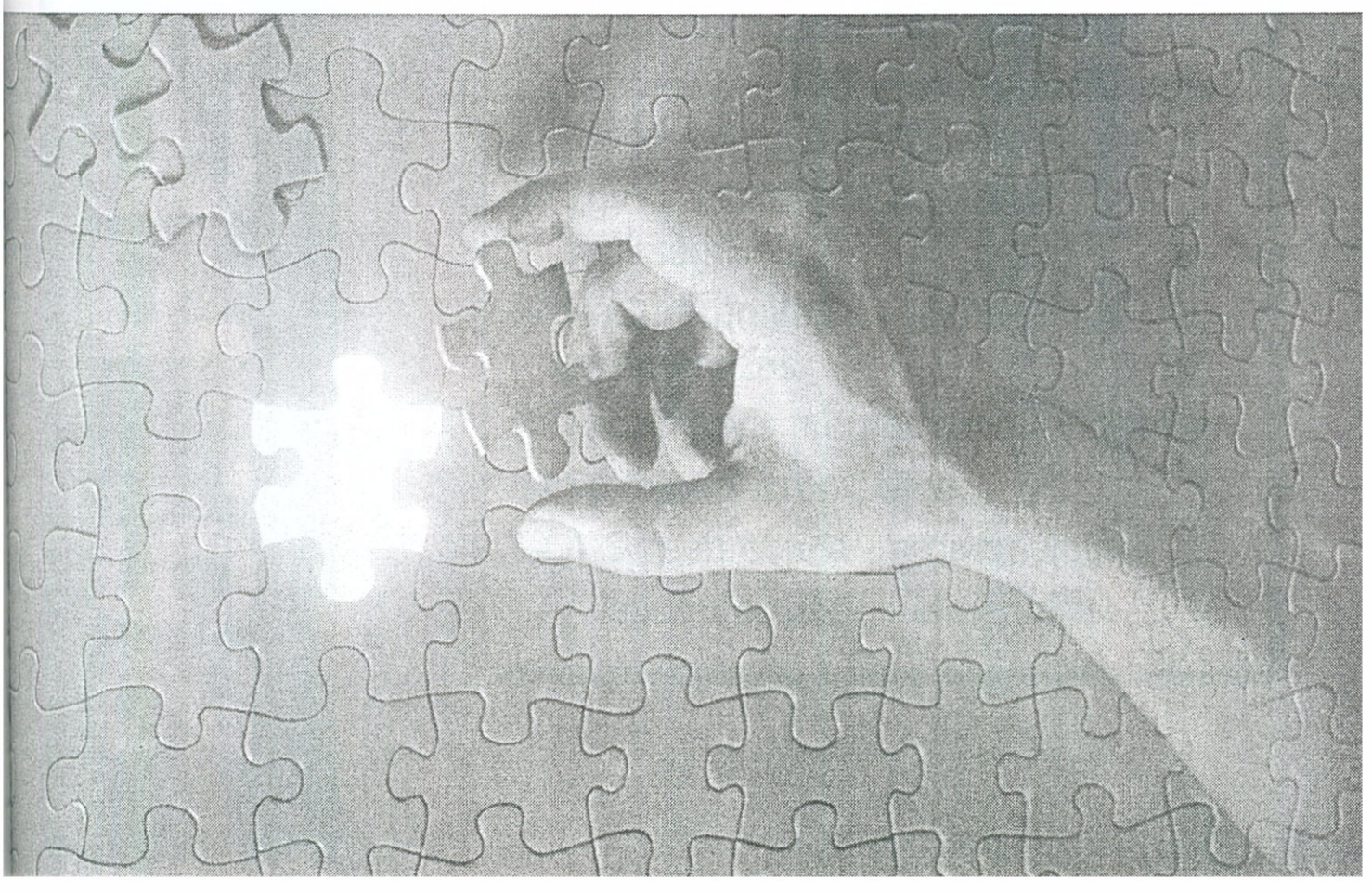
Ezzel a fejezettel a 3. rész végére értünk, amelyben főleg a komponensek fejlesztésével foglalkoztunk. Áttekintettük, hogyan kell Delphi komponenseket készíteni, hogyan kell a Delphi környezetet kiegészíteni, hogy az új komponenseket használatát könnyebbé tegyük. Továbbá hogyan készítsünk DLL fájlokat, hogyan helyezzünk osztályokat DLL fájlokba, hogyan írjuk COM osztályokat és interfészeket. A COM használatának olyan részeire is kitértünk, mint az OLE, OLE dokumentumok, ActiveX és ActiveFormok.

A kész XFMulti példaprogram: a Web böngészőben futó program szinte teljesen úgy néz ki, mint a 14. fejezetben megismert NoteBook program.

A könyv következő részében teljesen más témával foglalkozunk. A Delphi olyan részeiben fogunk elmélyedni, mint az alkalmazások szerkezete, párhuzamos programozás, hibakeresés, Windows erőforrások, fájlkezelés, nyomtatás, DDE és a multimédia.

IV. Rész

Fejlett Delphi programozás



25. FEJEZET

Az alkalmazások szerkezete

E fejezet tartalma:

- Az Application objektum használata
- Egy alkalmazás példányainak kezelése
- Komponensek nélküli programozás
- Események, üzenetek és párhuzamos programozás a Windowsban
- Egy óra elkészítése időzítő segítségével
- Háttérszámítás és párhuzamos programozás
- Párhuzamos programozás a Delphiben
- Szálak szinkronizálása
- A Screen objektum használata
- Állapotinformációk elmentése

Az alkalmazások szerkezete

A korábbiakban számos egy vagy több formból készített Delphi alkalmazást láthattunk. Minden alkalmazás formokból és a formokon elhelyezett komponensekből állt, a projektek forráskódjaiban azonban szerepel még egy elem: az Application (alkalmazás) objektum. Ennek a tizedik fejezetben bemutatott objektumnak hasonló a szerepe, mint a zenekar karmesterének; ugyanakkor ez is egy önálló ablak. Ebben a fejezetben az Application, és egy másik, az általunk már használt Screen objektum lehetséges felhasználási lehetőségeit ismerhetjük meg. A két objektum szerepének tárgyalása során több ide vonatkozó témáról ejtünk majd szót, beleértve a Windows üzenetek részletes tárgyalását, az időzítőket, a háttérben futó műveleteket, és a párhuzamos programozást. Szó lesz ezenkívül az INI fájlokról, a Registry-ről és társaikról.

Az Application objektum használata

Egy új, üres alkalmazás létrehozásakor a Delphi néhány új programsort ad a projekt fájlhoz, mely a következőt tartalmazza:

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Ez a kódrészlet a `TApplication` osztályból példányosított Application globális objektumot használja, ami a VCL Forms egységében van definiálva. Valójában ez is egy komponens, bár az Object Inspector segítségével nem állíthatjuk át a jellemzőit. Ez azonban nem igazán jelent problémát, mivel az alkalmazásnak kevés tulajdonsága van. Ezek a tulajdonságok az indítandó állomány neve, az alkalmazás megnevezése (alapértelmezés szerint az indítandó fájl neve, kiterjesztés nélkül), és néhány esemény. A Project Options párbeszédablak **Application** lapján beállíthatjuk a globális Application objektum néhány tulajdonságát.

Íme a `TApplication` tulajdonságainak és eseményeinek egy része (nem szerepelnek többek között a tippekre vonatkozó tulajdonságok, melyekkel a 12. fejezet részletesen foglalkozik):

- Az Active egy csak olvasható tulajdonság, mely azt jelzi, hogy az alkalmazás aktív-e.
- Az ExeName az indítandó állomány neve.
- A Handle az alkalmazás láthatatlan ablakához kapcsolódó kezelő.
- Az Icon a Tálcában használt ikon, mint azt a 27. fejezetben látni fogjuk.

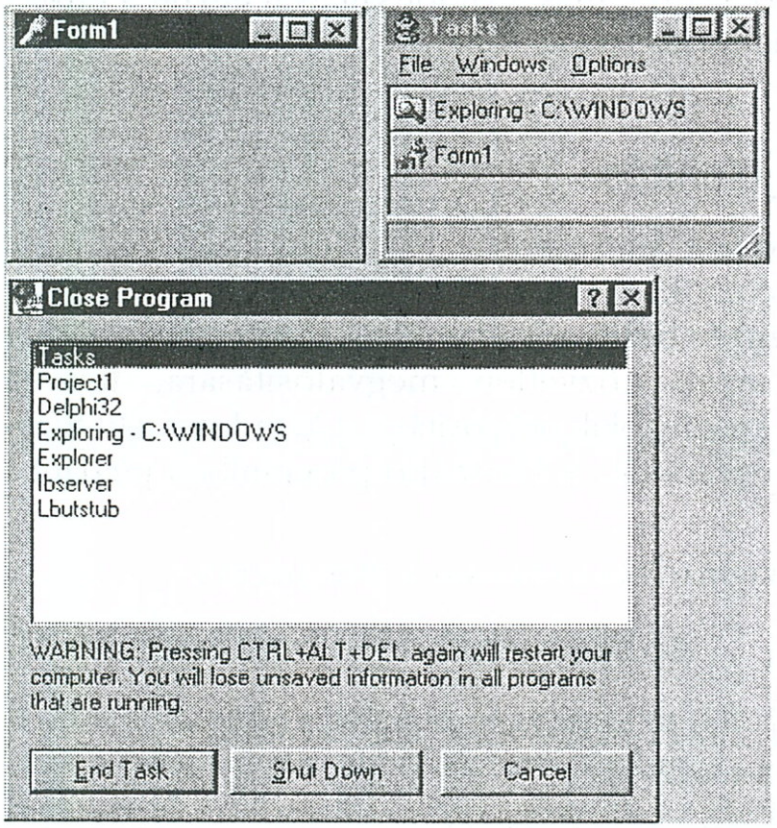
- A MainForm a Delphi program főformja.
- A ShowMainForm mondja meg, hogy indításkor a főform látszódjon-e.
- A Title az alkalmazásnak a Tálcán megjelenő megnevezését tartalmazza.

Az osztályhoz tartozó események:

- OnActivate: akkor következik be, ha a felhasználó aktiválja az alkalmazást azzal, hogy a fókuszt annak valamelyik ablakára helyezi át. Az alkalmazás és a form aktiválására hamarosan láthatunk majd egy példát.
- OnDeactivate: akkor következik be, ha egy másik Windows alkalmazás válik aktívvá.
- OnException: akkor következik be, amikor kivétel lép fel. Ezt az eseményt a 17. fejezetben (a DBError példában) használtuk.
- OnIdle: akkor következik be, ha az alkalmazás nem rendelkezik várakozó üzenettel. A fejezet későbbi részében egy példában használni fogjuk ezt az eseményt.
- OnHelp: akkor következik be, ha a felhasználó aktiválja a Súgó rendszert. A `TApplication` osztály HelpContext és HelpJump eljárásai automatikusan kiváltják ezt az eseményt.
- OnHint: egy tipp megjelenése előtt következik be, részletek a 12. fejezetben.
- OnMessage: akkor következik be, amikor az alkalmazás-ablakához egy Windows üzenet érkezik, ahogyan azt egy példában rövidesen látni fogjuk.
- OnMinimize: az alkalmazás kis méretűre állításakor következik be.
- OnRestore: az alkalmazás előző méretűre állításakor következik be.
- OnShowHint, egy tipp megjelenítésekor következik be, lásd a 12. fejezetet.

Az Application objektum szerepét először egy példán keresztül vizsgáljuk. Készítsünk egy új alkalmazást, fordítsuk le (a fájlokat az alapértelmezett UNIT1.PAS és PROJECT1.DPR néven elmentve), és futtassuk. Egy üres, Form1 fejlécű ablak és a hozzá tartozó ikon jelenik meg a Tálcán, az alkalmazás fő ablakának neve Project1. Ilyenkor valójában a Delphi nyit egy ablakot az Application objektum részére, és egy másodikat a fő ablak részére, amely az alkalmazás-ablakhoz tartozik. Az alkalmazás-ablaka tulajdonképpen nulla széles és nulla magas, így nem is látható (viszont nem rejtett, mert az befolyásolná a viselkedését).

Próbáljunk ki két további dogot. Mint azt a 25.1. ábra jobb felső részén is láthatjuk, az új Windows 95 **Futó programok** alkalmazásban (a WINDOWS könyvtárbeli TASKMAN.EXE) megnézhetjük a főablak nevét. Ebben az ablakban a program Form1-ként szerepel. Most a CTRL-ALT-DEL lenyomásával hívjuk elő a Windows 95 **Program bezárása** ablakát. Ahogyan a 25.1. ábra alján is látható, az ablak a futó alkalmazások, folyamatok listáját jeleníti meg, melyek mellett az új Delphi alkalmazásunkhoz kapcsolódóan szerepel a Project1 név is.



25.1. ábra *A Windows 95 Tasks alkalmazása kilistázza a fő ablakot, a Program bezárása ablaka pedig egy listát jelenít meg a folyamatokról, az éppen futó alkalmazásokról*

A Delphi futtatásakor a Windows 95 a Tálcán a Delphi alkalmazásnevét, míg a **Futó programok** ablakban és az ALT+TAB lenyomásakor megjelenő váltóablakban a főform nevét jeleníti meg. Nyilvánvaló, hogy a Delphi ugyanúgy működik, mint a vele készített alkalmazások (mivel magát a Delphit is Delphivel készítették). Ha a Delphi fő ablaka (amelyik a menüsort tartalmazza) látható, a projekt ablak címkéje jelenik meg, a Tálcán azonban a projektnév hiányzik. Ennek oka az, hogy a Tálcán nem a fő ablak, hanem az alkalmazás-ablak címkéje szerepel. Előny ez, vagy hátrány? Tapasztalatlan felhasználóknak a két különböző kifejezés zavaró lehet. Az alkalmazás nevének azonban megvan az az előnye, hogy állandó, míg a program címkéje az aktuális szerkesztett fájl nevének, vagy más dinamikus információnak megfelelően változhat.

Az alkalmazás nevének válasszuk a főform kezdeti nevét, vagy mindig ügyeljünk arra, hogy az alkalmazás és a form neve összhangban legyen. Ennek érdekében minden alkalommal, mikor a főform címkéje megváltozik, az alkalmazás nevét is meg kell változtatnunk. Amint módosítottuk a főform címkéjét, írjuk be a következő kódrészletet:

```
Application.Title := Caption;
```

Ugyanezt akár a program indulásakor is elvégezhetjük (a főform OnCreate eseményében). Az Application objektum más tulajdonságaival együtt a Project Options párbeszédablak **Application** lapján lehetőségünk nyílik a kezdeti alkalmazásnév beállítására.

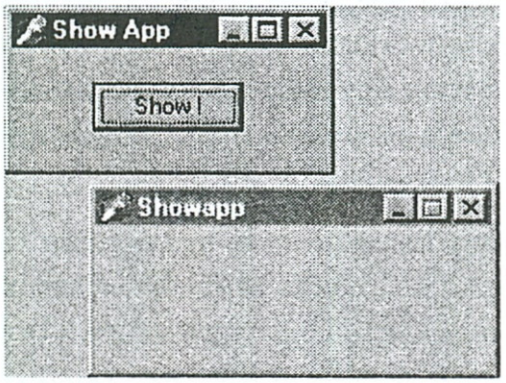
Az alkalmazás-ablak megjelenítése

Az Application objektum ablakának létezéséről a legkönnyebben úgy bizonyosodhatunk meg, ha megjelenítjük. Megjelenítenünk tulajdonképpen már nem is kell, elég csak átméretezni és beállítani a címke és a keret jellemzőit. A Delphiben nincs lehetőség ezek közvetlen megvalósítására, így az `Application.Handle` által kezelt ablak módosítását a Windows API függvényeinek segítségével kell elvégeznünk. A ShowApp programban például a következő kód szerepel:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldStyle: Integer;
begin
  // add border and caption to the app window
  OldStyle := GetWindowLong (
    Application.Handle, gwl_Style);
  SetWindowLong (Application.Handle, gwl_Style,
    OldStyle or ws_ThickFrame or ws_Caption);
  // set the size of the app window
  SetWindowPos (Application.Handle,
    0, 0, 0, 200, 100, swp_NoMove or swp_NoZOrder);
end;
```

Az ablakhoz kapcsolódó rendszerinformációkat a GetWindowLong és SetWindowLong API függvényeken keresztül érhetjük el. Esetünkben az ablak stílusának írására és olvasására a `gwl_Style` paramétert használjuk, amely magába foglalja a keretet, címet, rendszermenüt, keretikonokat, stb. A fenti kód lekérdezi a pillanatnyi stílusbeállításokat, és (az `or` utasítás segítségével) egy szabványos keretet és egy címkét ad a formnak. Az alacsonyszintű API függvények használatára a Delphiben ritkán van szükség, mert a `TForm` osztály néhány tulajdonságával kiválthatjuk alkalmazásukat. Jelen esetben azért kényszerültünk a kódot így megírni, mert az alkalmazás-ablak nem form.

A program hatására az alkalmazás-ablak a 25.2. ábrának megfelelően jelenik meg. Ajánlom, hogy egy kicsit játszadozzunk el ezzel a programmal. Bár saját programok készítésekor nem lesz szükségünk ehhez hasonló dolgok megvalósítására, a program futtatásával megismerhetjük egy Delphi program alkalmazás-ablakának és főablakának kapcsolatát. A Delphi alkalmazások belső szerkezetének tanulmányozása szempontjából ez egy nagyon fontos kiindulási pont.



25.2. ábra *A ShowApp programmal felfedett rejtett alkalmazásablak*

Az alkalmazások rendszermenüje

Az imént látotthoz hasonló nagyon ritka példáktól eltekintve a Tálcán mindig csak az alkalmazás-ablak látható. Ennek az ablaknak mindössze a rendszermenüjét aktiválhatjuk, ha rákattintunk az egér jobb gombjával. Mint már a 9. fejezetben is említettük, ha a rendszermenüről beszélünk, az alkalmazás és a főform menüje két különböző dolog. A 9. fejezet SysMenu példájában a főform rendszermenüjét különböző elemekkel bővítettük. A most következő SysMenu2 példában az alkalmazás-ablak Tálcán megjelenő rendszermenüjét szeretnénk testreszabni.

Első dolgunk az, hogy a program indulásakor új elemekkel kell bővítenünk az alkalmazás-ablakot. Íme a FormCreate eljárás kiegészített forráskódja:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // add a separator and a menu item to the system menu
  AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_SEPARATOR, 0, '');
  AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_STRING, idSysAbout, '&About...');
  // add the same items to the application system menu
  AppendMenu (GetSystemMenu (Application.Handle,
FALSE), MF_SEPARATOR, 0, '');
```

```
    AppendMenu (GetSystemMenu(Application.Handle, FALSE),
      MF_STRING, idSysAbout, '&About...');
    // handle application messages
    Application.OnMessage := AppMessage;
end;
```

A kód első része a főform rendszermenüjét egy elválasztóval és egy új elemmel egészíti ki. A másik két eljáráshívás ugyanezeket az elemeket az alkalmazás menüjéhez csatolja, egyszerűen az `Application.Handle`-re hivatkozva. Csupán ennyit kell tennünk a bővített rendszermenü megjelenítéséhez, amiről a program futtatásával meg is győződhetünk. A következő lépés az újonnan beépített elem kiválasztásának kezelése. Ehhez a fenti FormCreate eljárás utolsó sorához hasonlóan meg kell adnunk egy kezelőt az Application objektum OnMessage eseményének.

Ahogy azt a korábbi példákban láthattuk, a form üzeneteinek kezelésére új üzenetkezelő eljárásokat is írhatunk. Ugyanezt az alkalmazás-ablakkal viszont már nem tehetjük meg, egyszerűen azért, mert a `TApplication` osztályból történő öröklés meglehetősen összetett. Legtöbbször elég az osztály OnMessage eseményét kezelni, amely az üzenetsorból az alkalmazáshoz érkező minden egyes üzenetnél aktiválódik, amint azt a következő fejezetben részletesen tárgyaljuk.

Az OnMessage esemény kezeléséhez egyszerűen egészítsük ki a főformot egy újabb eseménnyel, adjuk meg a megfelelő paramétereket, telepítsük futásidőben, majd írjuk meg a kódot. Esetünkben elég csak a wm_SysCommand üzenettel foglalkozni, azzal is csak akkor, ha a wParam paraméter azt jelzi, hogy a felhasználó az imént beépített menüpontot, az idSysAbout-ot választotta:

```
procedure TForm1.AppMessage (var Msg: TMsg;
  var Handled: Boolean);
begin
  if (Msg.Message = wm_SysCommand) and
    (Msg.wParam = idSysAbout) then
  begin
    ShowMessage ('Mastering Delphi: SysMenu2 example');
    Handled := True;
  end;
end;
```

Az eljárás nagyon hasonlít ahhoz, amelyet a főform megfelelő rendszermenüpontjának kezeléséhez használtunk:

```
procedure WMSysCommand (var Msg: TWMSysCommand);
    message wm_SysCommand;
...
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
```

```
begin
  // handle a specific command
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu2 example');
  inherited;
end;
```

Alkalmazások és formok aktiválása

A formok és alkalmazások aktiválásának megértésére szolgál a szinte magyarázatra sem szoruló Activ nevű példa. A példa két formból áll, mindkettő rendelkezik egy Label (Label2) összetevővel az adott form állapotának megjelenítésére. A program e célra különböző színeket és szöveget használ, mint azt az első form OnActivate és OnDeActivate eseményeinek kezelői is mutatják:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Label2.Caption := 'Form2 Active';
  Label2.Color := clRed;
end;
procedure TForm1.FormDeactivate(Sender: TObject);
begin
  Label2.Caption := 'Form2 Not Active';
  Label2.Color := clBtnFace;
end;
```

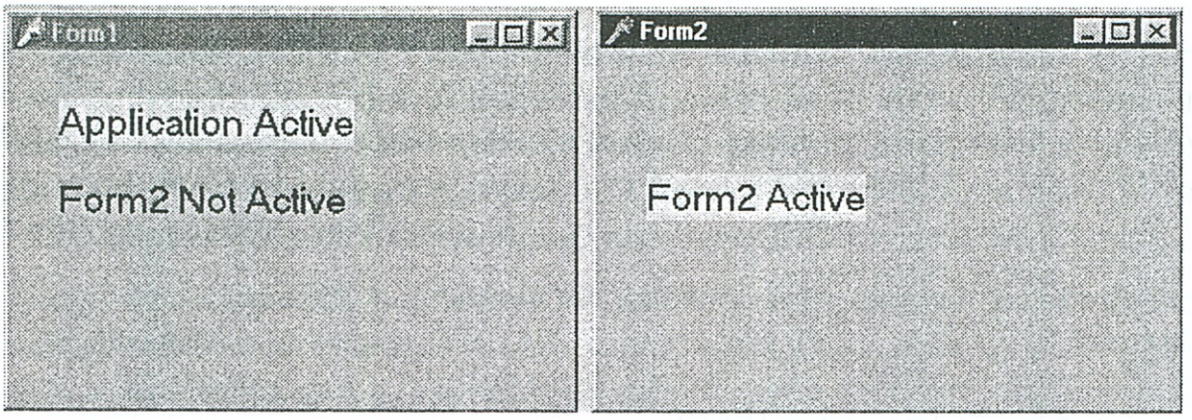
A második formban az előbbihez hasonló címke szerepel, hasonló kódrészlettel. A főform feladata az, hogy az OnCreate események kezelőjében az Application objektum két aktiváló eljárása részére két további kezelőt telepítsen:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnActivate := AppActiv;
  Application.OnDeActivate := AppDeActiv;
end;
```

A két eseménykezelő kódja nagyon egyszerű:

```
procedure TForm1.AppActiv(Sender: TObject);
begin
  Label1.Caption := 'Application Active';
  Label1.Color := clRed;
  Beep;
end;
procedure TForm1.AppDeActiv(Sender: TObject);
begin
  Label1.Caption := 'Application Not Active';
  Label1.Color := clBtnFace;
end;
```

A program futtatásakor láthatjuk, mikor lesz aktív az alkalmazás, és hogy ilyenkor formjai közül éppen melyik az aktív. A képernyőt (lásd a 25.3. ábrát) figyelve és az alkalmazás hangjait hallgatva minden Delphi által kiváltott esemény megfigyelhető. A program működésének megértéséhez azt ajánlom, mélyedjünk el egy kicsit a forráskódjában.



25.3. ábra *Az Active példa jelzi, hogy az alkalmazás aktív-e, és hogy melyik az alkalmazás aktív formja*

Egy alkalmazás példányainak kezelése

Az adott alkalmazás esetleges már létező példányainak ellenőrzését végző kód beépítése az egyik leggyakoribb változtatás, amelyet egy projekt fájlon a programozók végeznek. A Windows 3.1 programozók tudják, hogy egy rendszerváltozó, a hPrevInstance segítségével ez lekérdezhető. Ez egyike a szabványos WinMain függvény paramétereinek, amely függvény a C-ben írt hagyományos Windows programok belépési pontja. A 32-bites alkalmazásokban azonban ez a paraméter sajnos mindig 0.

A Delphi programozók négy globális változót használhatnak a Windows által minden egyes alkalmazásnak átadott WinMain paraméterek lekérdezésére:

- A HInstance az alkalmazás aktuális példányát képviselő belső Windows érték.
- A HPrevInst egy 16-bites alkalmazás előző példányának HInstance értékét tartalmazza, vagy egyetlen példány esetén a nullát. 32-bites alkalmazásnál ez az érték mindig nulla.
- A CmdLine az alkalmazásnak átadott parancssori paramétereket tartalmazza, mint például egy fájl neve.

- A CmdShow Windows 3.1 alatt egy megjelenítési kódot tartalmazott, jelezve, hogy az alkalmazás fő ablaka indításkor kis méretben, vagy teljes méretben jelenjen-e meg. Windows 95 alatt minden esetben egy rögzített érték kerül átadásra, a tényleges megjelenítési kódot pedig a GetStartupInfo API függvény meghívásával tudhatjuk meg.

Ez a négy paraméter jó szolgálatot tehet számunkra, mert nagyban elősegítik az alkalmazásnak a Windows környezetbe való zökkenőmentes beépülését.

A fő ablak egy másolatának felkutatása

Ha az alkalmazást egy példányban kívánjuk futtatni, akkor a projekt forráskódjának inicializáló részének valahogy így kell kinéznie:

```
var
  Hwnd: THandle;
begin
  Hwnd := FindWindow ('TForm1', 'One Copy');
  if Hwnd = 0 then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```

A FindWindow API függvény két paramétert vár: az ablakosztály nevét (a form ablaktípusának nyilvántartásához használt név, vagy a rendszerbeli WNDCLASS) és a keresett ablak címkéjét. A két paraméter közül az egyik szabadon elhagyható. Delphi alkalmazásokban a WNDCLASS ablak osztály neve megegyezik a form osztályának (például TForm1) Object Pascalbeli nevével. A FindWindow függvény visszatérési értéke a megfelelő ablakhoz tartozó kezelő, vagy nulla, ha nem található a paraméterekkel egyező ablak.

A fenti kódot használva a felhasználó csak akkor indíthatja el az alkalmazás egy új példányát, ha még egy sem létezik, ellenkező esetben semmi sem történik. A programnak a Delphi IDE környezetéből történő futtatása esetén sajnos már létezhet egy ugyanolyan címkéjű és osztályú ablak: ez a form tervezési időben használt példánya, következésképpen ilyenkor a program egyszer sem indul el. A form és a hozzá tartozó forrásállomány, vagy a projekt bezárása után a Windows Intézőből való futtatáskor azonban már elindul a program.

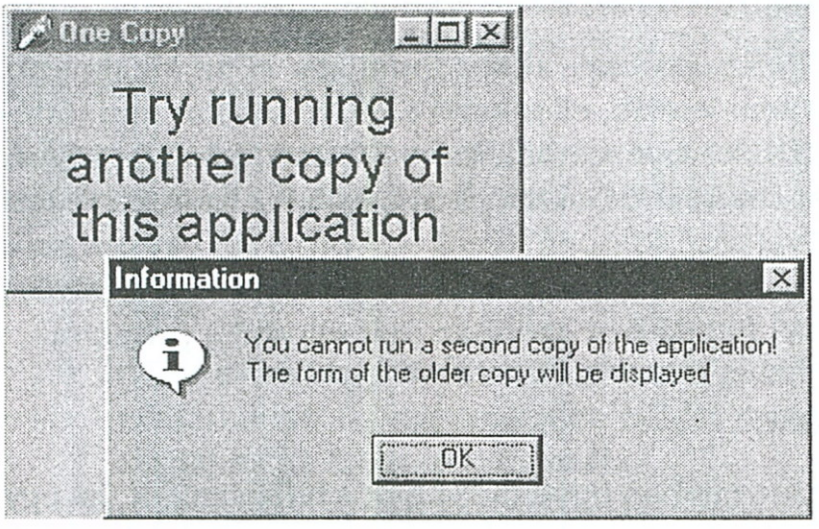
Az előző főform aktiválása

A program továbbfejlesztett változatában figyelmeztetjük a felhasználót, ha egy második példányt is elindít, majd aktiváljuk a program előző példányának főformját. A rendszer különböző kisebb alkalmazásaitól a nagy kereskedelmi alkalmazásokig számos Windows program így viselkedik, és ugyanez található a OneCopy1 példában is.

Az alkalmazás előző példányának aktiválására nem használhatjuk a Windows 3.1 programoknál népszerű SetActiveWindow API függvényt. Ehelyett az új SetForegroundWindow függvényt használjuk, amely még a más folyamatok által lefoglalt ablakokra is működik. Ezek szerint a fenti `if` utasítás mögé egy `else` ágat írhatunk:

```
var
  Hwnd: THandle;
begin
  Hwnd := FindWindow ('TForm1', 'One Copy');
  if Hwnd = 0 then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end
  else
  begin
    MessageDlg (
      'You cannot run a second
        copy of the application!' +
      #13+'The form of the older copy
        will be displayed', mtInformation, [mbOK], 0);
    SetForegroundWindow (Hwnd);
  end;
end.
```

A kód lefordítása előtt a projekt forráskódjának `uses` részét bővítsük ki a `Dialogs` és `Windows` egységekkel. A programnak a Delphi IDE környezetéből történő futtatásakor most is problémák lehetnek.



25.4. ábra *A OneCopy1 példa első és második példánya. Az OK-ra kattintva az első példány újra aktiválódik*

A OneCopy1 példa egy csupasz program, melynek főformja csak egy egyszerű címkét tartalmaz. Futás közben ez a 25.4. ábrán látható, ahol egy második példány került indításra. Az üzenetablakban az OK-ra kattintva az első példány aktiválódik, és a második befejezi futását.

Ha azt szeretnénk, hogy programunk az általános Windows-os szokásoknak megfelelően működjön, akkor egyszerűen csak el kell távolítanunk az üzenetet, és a korábbi példányt közvetlenül kell aktiválni. Vegyük észre azonban, hogy a program csak akkor működik, ha az alkalmazás előző példányának fő ablaka nem kis méretű. Ha az alkalmazás kis méretű, akkor a form ablaka rejtett, és az aktiváló kód hatástalan. Különböző függvények segítségével előhozhatjuk a rejtett formot, de ez sokkal több problémát okoz, mint amennyit megold. Végül is oda jutunk, hogy a form ablaka és a kis méretű alkalmazásablak egyszerre vannak jelen a képernyőn (ahogyan a ShowApp programban láthattuk).

Következő példánk nem elegáns, de működő módon oldja meg a problémát. Ez a technika messze nem tökéletes, inkább a felhasználó által definiált Windows üzenetek kezelését szemlélteti.

A felhasználó által definiált Windows üzenetek kezelése

A OneCopy2 programban arra kellene kérnünk egy másik alkalmazást, az előző példányt, hogy aktiválja magát, akkor is, ha nem látható. Mindez megoldható, ha a formnak küldünk egy felhasználó által definiált Windows üzenetet – egy olyan üzenetet, melyet a form egy általunk írt eljárás segítségével kezelni tud. Ellenőrizhetjük, hogy a form kis méretű-e, majd egy új üzenetet küldhetünk a korábbi ablaknak. Az utolsó szakaszban látott kódrészletet helyettesítő kód:

```
begin
  Hwnd := FindWindow ('TForm1', 'One Copy');
  if Hwnd = 0 then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end
  else
  begin
    if not IsWindowVisible (Hwnd) then
      PostMessage (Hwnd, wm_User, 0, 0);
    SetForegroundWindow (Hwnd);
  end
end.
```

A PostMessage API függvény az első paraméter által kijelölt célablakhoz tartozó alkalmazás üzenetsorának küld egy üzenetet. Ez az üzenet a form forráskódját egy speciális függvénnyel kibővítve kezelhető:

```
public
  procedure WMUser (var msg: TMessage);
    message wm_User;
```

Most már megírhatjuk az eljárás kódját, ami mellesleg elég egyszerű:

```
procedure TForm1.WMUser (var msg: TMessage);
begin
  Application.Restore;
end;
```

A Delphi IDE környezetéből ez a program szintén nem fog megfelelően működni, mivel a FindWindow hívás visszatérhet a Delphi Form Designere által megjelenített form kezelőjével is. A program ellenőrzése érdekében nem elég pusztán bezárni (vagy elrejteni) ezt a formtervező ablakot, hanem a megfelelő forrásállományt, vagy az egész projektet kell bezárni, majd a létrehozott programot önmagában kell futtatni. A következő szakaszban található ennek a programnak egy újabb változata, amelynél ez a probléma már nem merül fel.

Keresés az ablaklistában

Amint említettem, az utolsó példában használt FindWindow API függvény nem mindig megbízható. Nem tud különbséget tenni például az alkalmazás előző példányának formja, és a Delphi környezetében a tervezés közben használt form között. E problémák megoldására készült a példának egy más megközelítést használó harmadik, OneCopy3 nevű változata.

A rendszerben egy meghatározott főablak felkutatására az EnumWindows API függvények használhatók. Ezek a kereső függvények furcsák a Windowsban, mert ahogyan a 4. fejezetben láthattuk, általában egy másik függvényt várnak paraméterként. A kereső függvények egy függvénymutatót (ezt gyakran callback függvénynek nevezik) várnak paraméterül. Ez a függvény a lista (esetünkben a főablakok listája) egyes elemeire végrehajtásra kerül, míg a lista véget nem ér, vagy a függvény igaz értékkel nem tér vissza.

A program kezdeti részét (a FindWindow hívást) az új kóddal helyettesíthetjük. Íme a projekt inicializáló részének végleges változata:

```
var
  OldHwnd: THandle;
begin
  OldHwnd := 0;
  EnumWindows (@EnumWndProc, Longint (@OldHwnd));
  if OldHwnd = 0 then
  begin
```

```
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end
  else
  begin
    if not IsWindowVisible (OldHwnd) then
      PostMessage (OldHwnd, wm_User, 0, 0);
    SetForegroundWindow (OldHwnd);
  end;
end.
```

Az OldWin egy cím szerint átadott ablakkezelő változó, melyben a keresés eredményét fogjuk tárolni. Az EnumWndProc az egyes ablakok osztályainak nevét a TForm1 karakterlánc keresésével ellenőrző saját függvény. Az eljárás egy példány megtalálásakor visszaadja a pillanatnyi, és a keresett programmodul fájlnevét (ez az alkalmazás indítandó állományának neve). Ha ez is egyezik, biztosak lehetünk benne, hogy ugyanannak a programnak egy korábbi példányát találtuk meg. Íme a felsoroló függvény teljes forráskódja:

```
type
  PHWND = ^HWND;
function EnumWndProc (Hwnd: THandle; FoundWnd: PHWND):
  Bool; stdcall;
var
  ClassName, ModuleName, WinModuleName: string;
  WinInstance: THandle;
begin
  Result := True; SetLength(ClassName, 100);
  GetClassName(Hwnd,PChar(ClassName),Length(ClassName));
  ClassName := PChar(ClassName);
  if ClassName = 'TForm1' then
  begin
    SetLength (ModuleName, 200);
    SetLength (WinModuleName, 200);
    GetModuleFileName ( HInstance,
      PChar (ModuleName), Length (ModuleName));
    ModuleName := PChar(ModuleName); // adjust length
    WinInstance := GetWindowLong (Hwnd, gwl_hInstance);
    GetModuleFileName ( WinInstance,
      PChar (WinModuleName), Length (WinModuleName));
    WinModuleName:=PChar(WinModuleName); //adjust length
    if ModuleName = WinModuleName then
    begin
      FoundWnd^ := Hwnd;
      Result := False;
    end;
  end;
end;
```

Vegyük észre, hogy a függvényt `stdcall`-ként kellett definiálnunk, mivel mutatóként adjuk át egy másik függvénynek, majd a rendszer hívja meg. A Windows csak azokat a függvényeket tudja megfelelően meghívni, melyeket a Pascal kódban `stdcall`-ként megjelölünk, mert a rendszer nem érti az alapértelmezett Delphi hívásokat.

Kölcsönös kizárás (mutex) használata

Egy teljesen más megközelítési mód a mutex, avagy a kölcsönösen kizáró objektum (Mutual Exclusion Object) használata. Ez egy tipikus Win32 megközelítés, melyet, mint azt a fejezet későbbi részében látni fogjuk, szálak szinkronizálásánál is előszeretettel használnak. Két különböző alkalmazás, vagy (pontosabban) ugyanazon alkalmazás két példányának összehangolására most egy mutexet fogunk használni.

Ha egy alkalmazás elkészít egy adott nevű mutexet, a WaitForSingleObject Windows API függvény segítségével azután bármikor leellenőrizheti, hogy az objektumot birtokolja-e egy másik alkalmazás. Ha a mutexnek nincs tulajdonosa, a függvényt hívó alkalmazás lesz az. Ha a mutex már foglalt, az alkalmazás addig vár, míg a megadott várfakozási idő (a függvény második paramétere) le nem jár, majd hibakóddal tér vissza.

Az előző projekt forráskódját a OneCopy4 példában található alábbi kóddal helyettesíthetjük:

```
var
  hMutex: THandle;
begin
  HMutex := CreateMutex (nil, False, 'OneCopyMutex');
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut
then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```

A példa másodszori futtatásakor az alkalmazás egy újabb másolata átmenetileg megjelenik (az ikonja feltűnik a Tálcán), majd a megadott idő lejártával megsemmisül.

Komponensek nélküli programozás

Egy Delphi programozó munkájának nem kell komponensek használatára korlátozódnia. Igaz ugyan, hogy a komponensek központi szerepet játszanak ebben a környezetben, de a formokon kívül egyetlen összetevőt sem tartalmazó Delphi alkalmazások készítése is lehetséges (ezenkívül készíthető form

nélküli program is, de ez már egy nagyon speciális eset). Mely alkalmazások készíthetők el komponensek nélkül? A Windows alkalmazásokat eredetileg tiszta C nyelven írták, és a Delphiben is lehetőség van az alkalmazások hasonló alacsonyszintű megközelítésére.

A tényleges kérdés inkább az, hogy „Mely alkalmazásokat könnyebb komponensek nélkül elkészíteni?". Valójában csak nagyon kevés ilyen alkalmazás van. A komponensek felőli megközelítés kis és nagyobb alkalmazásoknak egyaránt javára válik. A Delphi előre megadott összetevői persze bizonyos szempontból megkötik a programozó kezét, a környezetet a megfelelő egyéni komponensekkel bővítve azonban bármilyen összetett alkalmazás elkészíthető.

A `TStreams`-hez vagy a `TLists`-hez hasonló Delphi objektumok még akkor is hasznosak lehetnek, ha a felhasználói felület szegényes, vagy éppen hiányzik, mint például képernyővédőknél, vagy eszközmeghajtóknál. Ebben a fejezetben például fel fogunk építeni egy analóg órát mutató összetett programot; a példa egy formon alapul, de a komponenseket csak nagyon korlátozottan használja, mindössze egy időzítő lesz majd benne.

A legkisebb Delphi program?

Mint az előző példákban láthattuk, egy alkalmazás projekt állományát bizonyos kódrészletekkel közvetlenül kiegészíthetjük. A projektek Delphiben történő módosítása egyaránt történhet a Project Manager használatával, a **Project Options** megfelelő beállításaival (ez az ajánlott), vagy a projekt forráskódjának közvetlen módosításával.

A projekt inicializáló részébe tetszőleges programrészt írhatunk. Egy form elkészítése és az alkalmazás futtatása helyett előidézhetünk egy csipogást, megjeleníthetünk egy üzenetablakot, vagy futtathatunk egy másik programot. A formok vagy az Application objektum használatának mellőzése azzal az előnnyel jár, hogy a futtatandó állomány mérete hihetetlenül lecsökken, mivel programunkba ilyenkor nem kell belefordítani a VCL könyvtárat.

Az ilyen programoknak kevés haszna van, néha azonban jól jöhetnek. Egyszerű konzol-alkalmazásokat készíthetünk például Delphiben. Néha szükségünk lehet fájlműveletek végzésére anélkül, hogy a felhasználót bármiről is tájékoztatnánk. Természetesen a hasonló „gyors és csúnya" programok elkészíthetők formokkal is, mivel a saját célra használt kis programoknál a fájl mérete nem igazán számít.

A következő példához hasonló programokat úgy is nevezhetnénk, mint a Delphivel lefordítható legkisebb programok, de igazából ezek nem számítanak valódi Delphi programoknak. Kis Object Pascalban írt Windows alkalmazások, melyeket a Delphi környezetében írtak és fordítottak. Az egyik legrövidebb

Delphivel elkészíthető program (a CD mellékleten Beep néven szerepel) a következő:

```
program Beep;

uses
  Windows;
begin
  MessageBeep (0);
end.
```

Az indítandó állomány mérete? Csak néhány kilobájt, szemben egy kisebb, futásidejű csomagokat sem használó Delphi alkalmazás átlagosan pár száz kilobájtos méretével. Hasznos a program? Aligha: csupán előidéz egy csipogást, és véget ér. Hasonló méretű program készítéséhez a Project Manager segítségével távolítsuk el a projekt minden formját, és a fenti Beep programhoz hasonlóan állítsuk be az alapértelmezett `uses` utasítást.

A parancssor olvasása

A legkisebb tényleg jelentős program egy egyszerű példa, ami a Windows parancssori paramétereit használja. Bár egy grafikus felhasználói felületben a felhasználók nagyon ritkán adnak meg parancssori paramétereket, a Windows parancssorában átadott paraméterek a rendszer számára fontosak. Ha például egy fájlkiterjesztés és egy alkalmazás társítva van, a programot egy társított fájl kiválasztásával egyszerűen futtathatjuk. Ha duplán kattintunk egy (BMP kiterjesztésű) bitkép állományon, a Paint rendszerint elindul, és automatikusan betölti az adott fájlt. Gyakorlatilag egy társítással rendelkező állományra duplán kattintva a megfelelő Windows program (például a Paint) elindul, és a kiválasztott fájlokat az alkalmazás a parancssorban kapja meg. Az már csak a programon múlik, hogy megnyitja-e a paraméterként kapott állományokat, de egy jól nevelt Windows alkalmazásnak illik ezt megtennie.

A következő StrParam (karakterlánc paraméterek) példa a parancssor (vagy egy string parancs) használatát mutatja be. Az alábbi utasítás a parancssor szövegét mutatja meg:

```
ShowMessage (CmdLine);
```

Indításakor láthatjuk, hogy Windows 95 (és Windows NT) alatt a parancssor első paramétere tartalmazza az indítandó állomány teljes elérési útját, majd a többi parancssori paraméter következik. A program elérési útja dupla idézőjelek között szerepel, így ha az elérési útra nincs szükségünk, csak a paramétereket szeretnénk megkapni, megkereshetjük a string második dupla idézőjel karakterét, és azután már csak a kívánt paraméterek következnek.

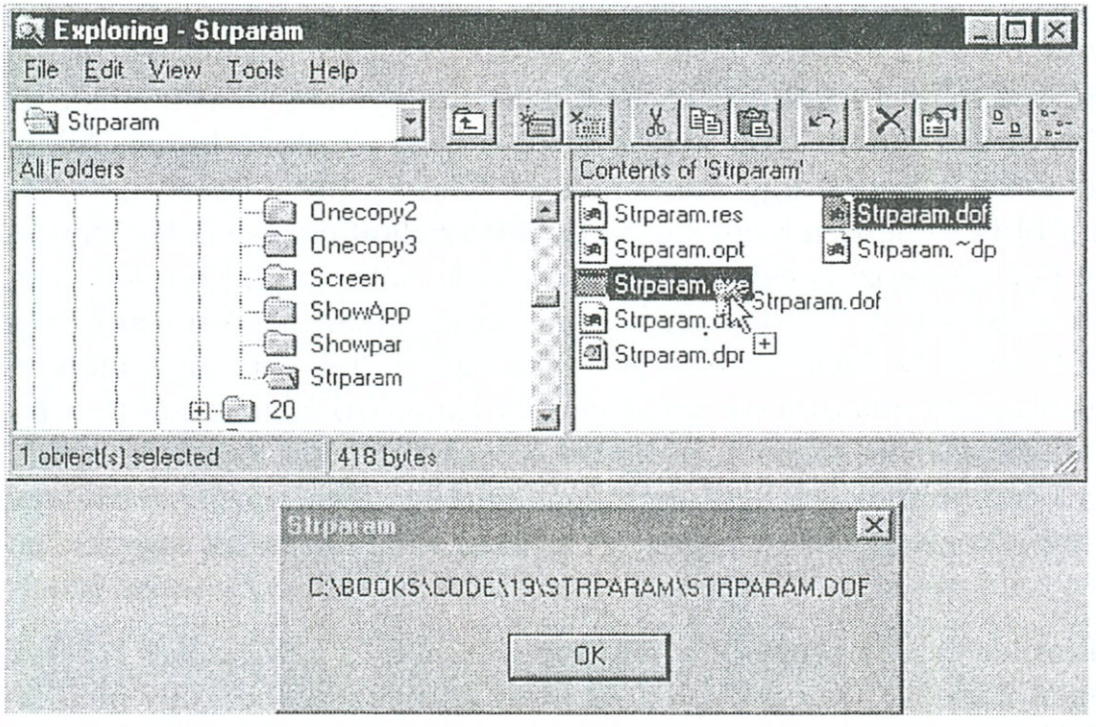
A probléma megoldásának egyik (de nem a legjobb) módja, ha beolvassuk a CmdLine karakterláncot, majd módosítjuk. A Delphi két egyszerű függvényt

nyújt a parancssori paraméterek kezelésére: ParamCount és ParamStr. Az első a paraméterek számával, a második pedig az adott helyen lévő paraméterrel tér vissza. Dupla idézőjelek közé zárva szóközt is tartalmazó paraméterek is átadhatók, mint például a hosszú fájl nevek, vagy elérési utak. A ParamStr(0) segítségével megkaphatjuk a program teljes elérési útját. Íme egy az előbbi függvényeket használó program:

```
begin
  if ParamCount > 0 then
    ShowMessage (ParamStr (1))
  else
    ShowMessage ('No command line');
end.
```

A StrParam példa forráskódjában megtalálható a kód mindhárom változata. A programot különféleképpen lehet ellenőrizni. Önmagában, parancssori paraméter nélkül futtatva a „No command line" üzenet jelenik meg. Ha hibakeresés közben parancssori paramétert szeretnénk átadni, használhatjuk a Delphi **Run | Parameters** menüpontját.

Egy másik módszer, hogy megnyitjuk a Windows Intézőt, belépünk a program indítandó állományát tartalmazó mappába, és egy másik fájlt ráhúzunk a programfájlra. A Windows Intéző elindítja a programot, a ráhúzott fájl nevét parancssori paraméterként átadva. A 25.5. ábrán a Windows Intézőt láthatjuk.



25.5. ábra *Ha a Windows Intézőben egy fájlt az indítandó állományra húzunk, egy parancssori paramétert küldhetünk a StrParam példának*

Események, üzenetek, és párhuzamos programozás a Windowsban

A Windows alkalmazások működésének megértése érdekében szánnunk kell egy percet annak tárgyalására is, hogy környezetünk hogyan támogatja a párhuzamos programozást. Tisztában kell lennünk az időzítők (és a Timer komponens) és a háttérfeldolgozás szerepével. Ezután térünk rá a Windows 95 párhuzamos programozására és ennek Delphi alkalmazásokba történő beépítésére.

Egyszóval mélyebbre kell ásnunk magunkat a Windows belsejébe és a párhuzamos programozás lehetőségeibe. Mivel ez a könyv a Delphi programozásról szól, ez a téma nem lesz túl részletes, de áttekintést nyújt a Windows (és különösen a Windows 95) programozásában kevésbé tapasztalt olvasóknak.

Eseménykezelt programozás

Az eseménykezelt programozás azon alapszik, hogy bizonyos események előre meghatározzák az alkalmazás folyamatvezérlését. A program idejének legnagyobb részében az ilyen eseményekre vár, és néhányukra megfelelő kóddal válaszol. Ha például a felhasználó kattint az egér valamelyik gombjával, egy eseményt vált ki. Ilyenkor az abban a pillanatban az egérmutató alatt lévő ablakhoz egy, az iménti eseményt jellemző üzenet érkezik. Az adott ablak eseményeire válaszoló programrész megkapja az eseményt, feldolgozza, majd ennek megfelelően visszajelez. Ha a program befejezte az esemény feldolgozását, visszaáll várakozó, vagy „tétlen" állapotba.

Ahogyan az iménti magyarázatból is kiderül, az események egymás utániak; az egyes események kezelésére csak akkor kerül sor, ha az előző befejeződött. Mikor egy alkalmazás éppen egy eseménykezelő kódot futtat (tehát nem vár eseményt), a többi eseménynek egy külön ennek az alkalmazásnak fenntartott üzenetsorban kell várakoznia (kivéve, ha a program több szállal rendelkezik, külön üzenetsorokkal). A Windows 16-bites változataiban a párhuzamosság korlátozott volt, mert az alkalmazások felváltva válaszoltak saját üzeneteikre. Mikor egy alkalmazás válaszolt az üzenetére és várakozó állásba jutott, a további üzenetek kezelésére jogosult programok listájában utolsó helyre került. A Windows 3.1-ben nem volt lehetőség megállítani egy alkalmazást, ha az egy összetett eseménykezelést hajtott végre, így a többi alkalmazásnak egyszerűen várni kellett.

Az eseménykezelés és az üzenetsor a Windows 95-nek és a Windows NT-nek szintén a magját képezik, de ezekben a 32-bites környezetekben egy meghatározott idő lejárta után a rendszer megszakítja az éppen aktuális alkalmazást, és azonnal átadja a vezérlést a lista soron következő elemének. A megszakított alkalmazás csak akkor folytatódik, ha már minden más alkalmazás sorra került.

Ezt a módszert preemptív többfeladatúságnak nevezik, és a Windows 3.1-ből nagyon hiányzott.

A korlátozott multitasking miatt a Windows 3.1 alkalmazások különféle technikákat alkalmaztak, hogy megpróbálják algoritmusaikat kisebb darabokra tördelni, majd azokat egy időben lefuttatni. Ezek közé a technikák közé tartoznak az időzítők és a háttérszámítás. Ezek még mindig hasznosnak bizonyulnak, így a következő szakaszokban részletesebben lesz szó róluk.

A Win32-ben a Windows 3.1-hez hasonlóan, ha egy alkalmazás megválaszolta az eseményeit és várakozó állásba állt, nem tudja visszaszerezni a vezérlést egészen addig, míg újabb üzenetet nem kap (hacsak nem rendelkezik több szállal). Az időzítők emiatt továbbra is léteznek (mellesleg 16-bit kompatíbilisek). Még egy utolsó megjegyzés – az eseményekkel kapcsolatban tudnunk kell, hogy a beviteli események (mint az egér vagy a billentyűzet használata) egy alkalmazás teljes üzenetfolyamatának csak egy kis hányadát teszik ki. A legtöbb üzenet a rendszer belső üzenetei vagy a különböző vezérlők és ablakok közötti üzenetek. Még egy olyan mindennapi beviteli művelet, mint az egérkattintás is tömérdek üzenetet eredményezhet, melyek többsége belső Windows üzenet.

Ez utóbbiról a Delphihez adott WinSight segédprogram használatával is meggyőződhetünk. A WinSight-ban válasszuk a Message Trace megjelenítését és állítsuk be, hogy minden ablak üzenete szerepeljen a megjelenítéskor. Menjünk a Start gombra, majd végezzünk pár hétköznapi műveletet az egérrel. Pár perc alatt üzenetek százaival találkozhatunk. A WinSight a folyamatos megfigyelése természetesen jelentősen lelassítja a Windows futását. Normális sebességnél az üzenetek sokkal gyorsabbak, mint ahogyan a WinSight futása mellett megjelennek.

A Windows üzenetkézbesítése

Még mielőtt belekezdenénk valódi példák tárgyalásába, az üzenetek kezelésének egy másik kulcselemével kell foglalkoznunk. A Windows két különböző módon küldhet üzenetet egy ablaknak:

A PostMessage API függvényen keresztül, amellyel az alkalmazás üzenetsorát egy üzenettel egészíthetjük ki. Az üzenet kezelése csak akkor történik, amikor az alkalmazás sorra kerül (azaz a vezérlést megkapja a rendszertől), és a korábbi üzeneteket már feldolgozta. Ez egy aszinkron hívás, mert nem tudhatjuk, hogy az üzenet valójában mikor érkezik meg. A fejezet korábbi részében a OneCopy2 példában már használtuk ezt az API függvényt.

A SendMessage API függvény használatával, mely az üzenetkezelő kódot azonnal elindítja. A SendMessage megkerüli az alkalmazás üzenetsorát, és üzenetét közvetlenül a cél ablaknak vagy vezérlőnek küldi. Ez egy szinkron hívás.

A függvénynek még visszatérési értéke is van, mégpedig az üzenetkezelő kód által visszaadott érték.

Az üzenetek küldésének két módja közötti különbség hasonlít egy levél postai úton való elküldése, melynek során a kézbesítés némi időbe telik, és a fax küldése közötti különbségre, melynél a címzett azonnal megkapja az információt. Bár a Delphiben csak nagyon ritkán van szükség az ilyen alacsonyszintű függvények használatára, azért nem árt elgondolkodni, hogy egy ilyen típusú kódnál melyiket kellene használni.

A SendMessage használatának előnye nyilvánvaló: a rendszer feletti irányítást teszi lehetővé. Általában azonban egy nagyon egyszerű okból a PostMessage ajánlott: ha az üzeneteket a megfelelő sorok végére küldjük, minden futó alkalmazásnak nagyobb esélye van megkapni a vezérlést, így az egész rendszer megbízhatóbban működik. A Delphi tartalmaz egy speciális Perform eljárást, melynek hatása nagyon hasonlít a SendMessage-ére.

Egy óra elkészítése időzítő segítségével

Az eseménykezelt környezetekben felmerülő problémákat igen könnyen megismerhetjük egy órát megvalósító program elkészítése során. Egy órának az idő múlásával automatikusan frissítenie kell a kimenetét. Erre egy hagyományos megoldás az idő folyamatos ellenőrzése a rendszer órájában tárolt érték olvasásával. Bár egy ilyen program kifogástalanul működik, túl sok processzoridőt használ, így csökkenti a más alkalmazásoknak jutó erőforrásokat.

Lehetőség van a rendszeróra körülbelül egy másodpercenkénti olvasására, majd amint lehet, visszaadni a vezérlést a rendszernek. Hogyan ébred azonban alkalmazásunk újra, ha az egyes másodpercek elteltek? Nos, ennek biztosítása lesz az időzítő feladata. A Delphiben egy Timer összetevő egy meghatározott időtartam minden egyes elteltével egy OnTimer nevű eseményt kap.

Mielőtt folytatnánk az óra program példáját, figyeljük meg a Windowson belüli időzítő viselkedését különböző szempontokból (korábban már használtam ezt a komponenst, de nem magyaráztam el a viselkedését). Az időzítők alapja egy a rendszer órája által vezérelt megszakítás, mely pontosan meghatározott gyakorisággal keletkezik.

A probléma az, hogy más üzenetekhez hasonlóan az időzítő üzenetek is egy ablakhoz érkeznek, és az ablakot birtokló alkalmazás üzenetsorának végére kerülnek. Ez azt eredményezi, hogy az időzítő üzenetek nem a megfelelő ütemben érkeznek az alkalmazáshoz. Ha egy program kis időre lefoglalja a CPU-t (például egy fájl betöltésekor), az időzítő üzenetek változatlanul érkeznek, de míg a program vissza nem kapja a rendszertől a vezérlést, nem kerülnek feldolgozásra.

Ebben az esetben egy második probléma is felmerül. Mivel az időzítő üzenetek nagyon gyakoriak lehetnek (elméletileg minden ezredmásodperc; gyakorlatilag legfeljebb minden 60. ezredmásodperc), már nem férnek be az üzenetsorba. Ha egy újabb időzítő üzenet érkezik a sorba, és egy ugyanolyan üzenet már ott van, az újabb felülírja az előzőt. Ez azt jelenti, hogy egy alkalmazás a kapott időzítő üzenetek számából nem tudja pontosan megállapítani a közben eltelt időt. Ebből kifolyólag egy időzítő üzenet beérkezésekor biztosak lehetünk ugyan abban, hogy eltelt valamennyi idő, de nem tudhatjuk, hogy mennyi.

Hogyan használhatunk időzítőt egy óra elkészítéséhez? Egyszerűen helyezzünk a formra egy Timer és egy Panel összetevőt. Állítsuk az időzítő időtartamát 1000 ezredmásodpercre (azaz egy másodpercre), válasszunk a panel címkéjének egy alkalmas betűkészletet, és igazítsuk a panelt a form kliens területéhez. A form megépítése után a Timer komponens OnTimer eseményének kezelésére írjuk be a következő kódot:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Panel1.Caption := TimeToStr (Time);
end;
```

A Time függvény egy a pillanatnyi időt tartalmazó `TDateTime` objektummal tér vissza, melyet a TimeToStr függvény segítségével karakterlánccá alakítunk. A kód eredménye a 25.6. ábrán látható.

25.6. ábra *A Clock1 példa futás közben*

Amint a 4. fejezetben láthattuk, a TimeToStr függvény által visszaadott string formátuma néhány, a SysUtils egységben definiált változó értékétől függ. Ezeket az értékeket a Windows nemzetközi beállításai határozzák meg. Egy egyszerű teszt kedvéért indítsuk el a Clock1 példát, nyissuk meg a Windows Vezérlőpult-ját, válasszuk ki a Területi beállítások-at és állítsunk be a sajátunktól eltérő országot (vagy menjünk az Idő lapra és ott változtassunk pár értéket).

Alkalmazzuk az új beállításokat, és észrevehetjük, hogy a Clock1 példa automatikusan az új beállításokhoz igazodott.

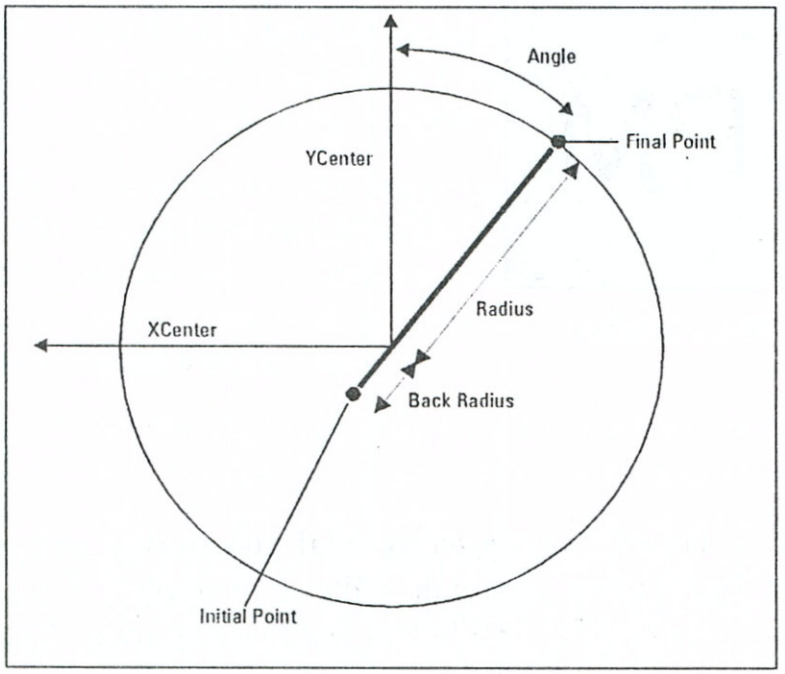
Ahogyan a példában is látható, az időzítők időfüggő alkalmazások írására alkalmasak. Az időzítő használata akkor is ajánlott, ha az alkalmazásnak egy bizonyos értéket kell figyelnie. Erre jó példa a következő fejezetben bemutatott Mem program. Néha időzítőket használhatunk a háttérben folyó számítások megvalósítására. A fejezet későbbi részében a Clock példa további finomításai után ezt az ötletet is megvizsgáljuk.

Egy grafikus óra

Most, hogy az eredeti ötletet megvalósítottuk, gondolkodhatunk a Clock példa új elemekkel való bővítésén. A digitális órák nem tartoznak a kedvenceim közé. Milyen lenne egy analóg óra csinos óramutatókkal? A Windowsban van már ugyan egy óra, és valójában nem lenne szükség egy új készítésére, de mindenkinek megvan a kedvenc órastílusa, és Windows óraalkalmazások tucatjai állnak rendelkezésre. Így hát egy újabb óra következik a Windowshoz.

Az új példa alapja a már felépített óra program. A mutatókkal rendelkező óra formja még ennél is egyszerűbb. Csak egy időzítő szerepel benne, panel nélkül. A kód legnagyobb része a form OnPaint eljárásában szerepel. Mivel három különböző óramutatót kell rajzolnunk, a TForm1 osztályt kibővíthetjük egy általános eljárással. Az eljárás neve DrawHand, és öt paraméterrel rendelkezik:

```
procedure DrawHand (XCenter, YCenter, Radius,
  BackRadius: Integer; Angle: Real);
```



25.7. ábra *A Clock2 példa DrawHand eljárásának kimenetének egy grafikus ábrázolása*

Az első két paraméter az óra közepének x és y koordinátáját jelenti, majd az óramutató mérete (azaz az általa rajzolt kör sugara), és a középponton keresztül a másik irányba való túlnyúlás nagysága következik. Az utolsó paraméter a mutató pillanatnyi szögét jelzi. Az öt paraméter szerepének rajzi szemléltetése a 25.7. ábrán található.

Az óramutatók analóg ábrázolását csak trigonometrikus függvények (szinusz és koszinusz) segítségével lehetett megoldani, ezért az Angle egy valós szám, nem pedig egész. A dolgok bonyolultabbá tétele érdekében középről való kiindulás helyett egy mutató az óra középpontján keresztül az ellenkező oldalra is túlnyúlhat, amint azt a 25.8. ábra mutatja. Íme a DrawHand függvény teljes forráskódja:

```
procedure TForm1.DrawHand (XCenter, YCenter,
  Radius, BackRadius: Integer; Angle: Real);
begin
  Angle := (Angle + 3*Pi/2);
  Canvas.MoveTo(XCenter -Round (BackRadius*Cos(Angle)),
    YCenter - Round (BackRadius * Sin (Angle)));
  Canvas.LineTo (XCenter + Round (Radius * Cos(Angle)),
    YCenter + Round (Radius * Sin (Angle)));
end;
```

Az Angle értékének kezdeti megváltoztatása azt eredményezi, hogy a kezdeti szög a vízszintes tengely helyett a függőleges tengelyre legyen relatív. A szöget háromnegyed fordulattal előre kell állítani, és mivel a teljes fordulat 2×Pi, a megfelelő kifejezés 3×Pi/2 lesz.

A FormPaint eljárásban a program megállapítja a form közepét és az óra sugarát, majd szögük kiszámítása után megrajzolja a mutatókat. A formot az Hour (óra), Minute (perc), és Second (másodperc) privát mezőkkel kellett bővíteni, melyek értékét a Delphi System egységében definiált DecodeTime eljárás segítségével az OnTimer függvény számítja ki. Az OnTimer a képernyőn lévő kép frissítése céljából meghívja a Refresh eljárást:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  HSec: Word;  {temporary value, not used}
begin
  {get the system time}
  DecodeTime (Time, Hour, Minute, Second, HSec);
  Refresh;
end;
```

A FormPaint eljárás egy részlete:

```
{1. Draw the minutes hand: Blue thick pen}
  Canvas.Pen.Width := 2;
  Canvas.Pen.Color := clBlue;
```

```
Angle := 2 * Pi * Minute / 60;
DrawHand (XCenter, YCenter,
  Radius * 90 div 100, 0, Angle);
{2. Draw the hours hand: Percentage of minutes
 added to hour to move the hand smoothly: Same pen
 as the minutes }
Angle := 2 * Pi * (Hour + Minute / 60) / 12;
DrawHand (XCenter, YCenter, Radius * 70 div 100,
  0, Angle);
```

25.8. ábra *A Clock2 program kimenete. Vegyük észre, hogy a másodpercmutató átlóg a középpont másik oldalára*

A FormPaint eljárás először kiszámítja a form közepét és meghatározza az óra sugarát, melyet a Radiusban tárol (a form újabb privát mezője), majd minden mutatót megrajzol. Először beállítja a mutató színét és méretét (a Canvasban kiválasztott Pen tulajdonságainak módosításával), majd az 1-től 60-ig terjedő számokat (perc és másodperc) radiánná alakítva (ezért van szükség a Pi-re) kiszámítja a szöget.

A kismutató megrajzolását végző kód valamivel összetettebb. A kismutató mozgásának érdekében a programnak az óra értékéhez hozzá kell adnia a következő órából eltelt idő százalékát (azaz a percek hatvanad részét). A rajzoló eljárás eleje emellett megjelöli az egész órák pozícióit, könnyebbé téve ezzel az óra leolvasását. A vonások helyét a program egy `for` ciklusban számítja ki, szintén a szinusz és koszinusz függvények segítségével (amint a CD mellékleten szereplő teljes forráskódban is látható).

A programban két érdekesebb rész található. Valahányszor a formot átméretezzük, a Refresh eljárást meg kell hívni, hogy újra fesse a teljes formot, az órát középre helyezze, és kiszámítsa az óra új sugarát. A másik furcsaság az OnCreate eseményben szereplő Timer1Timer eljáráshívás:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Timer1Timer (self);
end;
```

Ezzel a hívással a form elkészülése után azonnal megkapjuk a pillanatnyi időt. A Windows nem sokkal az OnCreate esemény után meghívja az OnPaint eseményt. Ha az idő ekkor még nincs kiszámolva, a program kimenete az első időzítő-eseményig értelmetlen lesz.

A másodpercmutató megrajzolása rasztermű veletekkel

A Clock2 program működik ugyan, de messze van a tökéletestől. Az óra képe bizonytalan, és sokat villog. Valójában a form teljes felülete másodpercenként letörlődik, majd a semmiből újraépül. A másodpercmutatónak mozognia kell, így ez látszik az egyetlen lehetséges megoldásnak, de ez nincs így. Az egész kép letörlése helyett elég csak az előző másodpercmutatót letörölni, majd újrarajzolni.

A probléma most az, hogy hogyan töröljük le a másodpercmutatót. A háttér színével húzhatnánk rá egy vonalat, de ezzel esetleg a másik két mutató, vagy a vonások bizonyos részeit is letörölnénk.

A Windows rasztermódjai

A megoldás a Windows rasztermű veleti módjaiban van, melyeket a Delphiben a toll objektumok Mode tulajdonsága jelez (a 14. fejezetben a Split4 példájában ugyanezt a megközelítést alkalmaztuk). A rasztermű veleti kódok segítségével szabályozhatjuk a tollak rajzoló függvények közbeni viselkedését. Néhány példa a rasztermű veleti kódokra:

- `pmBlack`, a beállított színtől függetlenül feketével rajzol
- `pmNot`, megfordítja a képernyő színét, szintén figyelmen kívül hagyva a beállítást
- `pmCopy`, az alapértelmezett beállítás, a toll színét a képernyőre másolja

Az egyes módok eredménye egyaránt függ a toll és a képernyő pontjainak aktuális színétől. A színek keverhetők, megfordíthatók, egy adott színnel maszkolható a másik, kizáró vagy (xor) végezhető a két szín között, stb.

Az óra példa másodpercmutatójának megrajzolásához szükséges rasztermű veleti mód a `pmNotXor`. Eredménye a két szín közötti kizáró vagy művelet

elvégzése utáni szín fordítottja. Fő tulajdonsága ennek a rajzolómódnak, hogy ha ugyanazt a vonalat kétszer rajzoljuk meg, eltűnik. Emellett színesben is rajzolhatunk. Pontosan ezt kerestük, mivel az új vonal rajzolása előtt ennek segítségével letörölhető a régi.

Megjegyzés

A `pmNotXor` mód egy kicsit valóban rejtélyes, nekünk azonban nem kell ténylegesen értenünk, hogy a színes képpontok hogyan keverednek. A lényeg, hogy a mód azt csinálja, amire szükségünk van.

A grafikus óra új változata

Az egyes időzítő üzenetek érkezésekor a másodperceket és a perceket mindenekelőtt el kell tárolnunk a form két új privát mezőjében: amiket OldMinute-nek és OldSecond-nek neveztem el. Ha a percek értéke nem változik, a másodperc korábbi értékét használva letöröljük az előző mutatót, majd kirajzoljuk az újat:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  HSec: Word;  {temporary value, not used}
begin
  {store the old values and get the system time}
  OldMinute := Minute;
  OldSecond := Second;
  DecodeTime (Time, Hour, Minute, Second, HSec);
  {if minutes haven't changed, move the
  seconds hand else redraw the whole clock}
  if Minute = OldMinute then
    DrawSecond
  else
    Refresh;
end;
```

A DrawSecond eljárás a mutató korábbi és új helyzetének megfelelően két vonalat rajzol:

```
procedure TForm1.DrawSecond;
var
  Angle, OldAngle: Real;
begin
```

```
    {delete the old line, drawing over it again}
    OldAngle := 2 * Pi * OldSecond / 60;
    DrawHand (XCenter, YCenter, Radius,
      Radius * 30 div 100, OldAngle);
    {draw the new line}
    Angle := 2 * Pi * Second / 60;
    DrawHand (XCenter, YCenter, Radius,
      Radius * 30 div 100, Angle);
end;
```

A kód helyes működéséhez a rasztermódot pmNotXor-ra kell állítani. Ha ezt a rajzoló kód végére írjuk, elkerülhetjük, hogy a másodpercmutató minden egyes megrajzolásakor be kelljen azt állítani. Íme az eljárás módosított változatának részlete (a Clock3 példa forráskódjából kivágva):

```
{3. Draw the seconds hand}
{Red thin pen, with 'not xor' raster mode}
Canvas.Pen.Width := 1;
Canvas.Pen.Color := clRed;
Canvas.Pen.Mode := pmNotXor;
Angle := 2 * Pi * (Second + 45) / 60;
DrawHand (XCenter, YCenter, Radius,
  Radius * 30 div 100, Angle);
```

Az ismételt függvényhívások elkerülésével növelhetjük a program sebességét. Az új, Clock3 nevű legújabb változat kimenete nagyon hasonló az előzőéhez, viszont már nem villog. Ezenkívül van egy másik különbség is: ha az óramutatók átfedik egymást, a rasztermód kombinálja a két mutató színét. Ezt nehéz egy fekete-fehér könyvben szemléltetni, ezért a két programot ajánlott inkább a képernyőn összehasonlítani.

Háttérszámítás és párhuzamos programozás

Az időzítők Windows alkalmazásokban való használatának egy másik célja a háttérszámítás valamilyen formájának megvalósítása. Tegyük fel például, hogy egy időigényes algoritmust kell megírnunk. Ha az algoritmust egy eseményre való válaszként írjuk meg, az alkalmazás az algoritmus futásának ideje alatt áll. A felhasználót a homokóra kurzor segítségével tájékoztathatjuk arról, hogy valami történik, de ez nem egy felhasználóbarát megoldás.

Ennél valamivel jobb ötlet, ha az algoritmust kisebb darabokra vágjuk, és az egyes darabok végrehajtása között lehetőséget adunk az alkalmazásnak megválaszolni az elintézetlen üzeneteit. Radikálisabb megoldás érhető el a Windows 95 és NT párhuzamos programozásának kihasználásával: arra is van lehetőségünk, hogy elindítsunk egy külön szálat az algoritmus részére, az alkalmazás pedig ezzel egy időben feldolgozhatja a beérkező üzeneteket.

Először megnézünk pár technikát a háttérszámítás egy szálon történő megvalósítására. A Windows 3.1-ben szükség volt ezekre a megoldásokra, és a Win32 alkalmazásokban szintén hasznosak. A továbbiakban feltárjuk a Delphi párhuzamos programozást támogató részeit és a háttérszámításra vonatkozó előnyeit, beleértve néhány összehangoló technikát, és a szálak prioritásának beállítását.

Háttérszámítás

Amint a következő példából kiderül, a háttérszámítás megvalósítására különböző módszerek vannak. A két legelterjedtebb ezek közül:

- Az Application.ProcessMessages függvény rendszeres meghívásával az esetleges várakozó üzenetek feldolgozásra kerülnek.
- A program egy lépésének végrehajtása valahányszor az Application objektum az OnIdle eseményt kapja.

Az Application objektumhoz kötődő két megközelítés a pillanatnyi foglaltságtól függően a programnak több vagy kevesebb rendszeridőt ad. A ProcessMessages hívása és az OnIdle események abban különböznek, hogy az előbbi hívásakor a kód több futásidőt kap, mint az OnIdle megközelítésnél. A ProcessMessages hívásával a program számolása közben lehetővé tehetjük a rendszernek más műveletek elvégzését, míg az OnIdle esemény használatával a rendszer háttértevékenységet folytathat, ha nincs elintézetlen feladata.

A háttértevékenység egy harmadik, kevésbé elterjedt (és összetettebb) megvalósítása szerint az alkalmazás a háttérművelet minden lépése után egy felhasználó által definiált üzenetet küld önmagának:

```
PostMessage (Handle, wm_User, 0, 0);
```

Az alkalmazás kis idő múlva megkapja az üzenetet, és végrehajthatja a következő lépést, majd újabb üzenetet küld önmagának, egészen addig, míg a háttértevékenység be nem fejeződik.

Prímszámok keresése

Az előző néhány lehetőség szemléltetése céljából készült a BackProc (háttértevékenység) példa. A program formja négy gombot tartalmaz a különböző technikák elindítására, négy folyamatsávot az algoritmusok aktuális helyzetének megjelenítésére, és négy címkét a végeredmény közlésére. Mindezek felett található a formon egy SpinEdit vezérlő, mellyel az algoritmus iterációinak számát változtathatjuk. A négy gomb valamelyikére kattintva a program megszámolja, hogy hány, a fenti SpinEdit vezérlő által meghatározott számnál kisebb prímszám található. Az egyes gombok különböző algoritmust jelentenek. A SpinEdit1 értékének módosításával számítógépünk sebességéhez mérten befolyásolhatjuk a számítás időtartamát.

A prímszámok hatékony keresésére számos nagyon intelligens megoldás létezik, itt azonban a háttérszámítás szemléltetéséhez egy valóban időigényes feladatra volt szükség, ezért a prímszámokat előállító függvény egyszerűen végigosztja a számot 2-vel, 3-al, és így tovább egészen a számot megelőző értékig.

Az IsPrime függvény a mod operátor segítségével megvizsgálja az egyes egész osztások maradékát, és ha az nulla, a szám nem prím. Ebben az esetben kilép a ciklusból és False-t ad vissza:

```
function IsPrime (N: LongInt): Boolean;
var
  Test: LongInt;
begin
  IsPrime := True;
  for Test := 2 to N - 1 do
  begin
    if (N mod Test) = 0 then
    begin
      IsPrime := False;
      break; {jump out of the for loop}
    end;
  end;
end;
```

Ezzel a függvénnyel már egy egyszerű ciklus elég ahhoz, hogy megszámoljuk hány prímszám található egy bizonyos határ alatt:

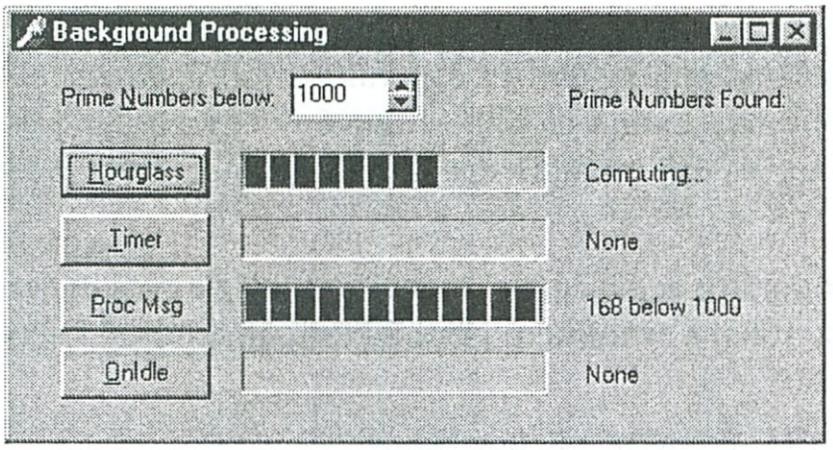
```
for Number := 2 to Max do
  if IsPrime (Number) then
    Inc (NPrimes);
```

A ciklus végén a végeredményt az NPrimes változó tartalmazza. A program egyszerű, de nem túl hatékony. Ugyanez az algoritmus négyszer szerepel, apró változtatásokkal a `for` ciklus kódjában.

A „homokórás" megközelítés

Az első, Hourglass feliratú gomb a legegyszerűbb megközelítést használja. Végigszámolja az algoritmust, tart amíg tart, anélkül, hogy a vezérlést átadná a rendszernek. A kurzor homokórára vált, és a folyamatsáv rendszeresen frissítődik, így a felhasználó látja, hogy történik valami. A 25.9. ábra a BackProc példa Hourglass gombjának lenyomása utáni kimenetet mutatja.

A program a homokórát a Screen objektum Cursor tulajdonságát crHourglass-ra állítva jeleníti meg. A végén vigyázni kell a normál kurzor visszaállításával, egy `try-finally` blokk finally részébe tegyük ezt a műveletet. Ez a megközelítés biztosítja a kurzor megfelelő visszaállítását, még akkor is, ha a kódban kivétel történt.



25.9. ábra *A BackProc példa az első gomb lenyomása után*

Az algoritmus megkezdése előtt két másik művelet történik. A jobb oldali címke Computing-ra vált, és a SpinEdit vezérlő értéke a Max változóba kerül. Ez a változó az ezt követő kódban többször szerepel, ezért a program először kiolvassa ezt az értékét, és eltárolja. Ez amellett, hogy a kódot hatékonyabbá teszi, a későbbi verziók miatt szükséges is, mert azok a ciklus ideje alatt más műveletek végrehajtását (a szerkesztőmező értékének megváltoztatását is beleértve) is megengedik. Íme az első változat teljes kódja:

```
procedure TForm1.HourButtonClick(Sender: TObject);
var
  Number, NPrimes, Max: LongInt;
begin
  Screen.Cursor := crHourglass;
  try
    NPrimes := 0;
    Max := SpinEdit1.Value;
    HourLabel.Caption := 'Computing...';
    for Number := 2 to Max do
    begin
      if IsPrime (Number) then
        Inc (NPrimes);
      ProgressBar1.Position := Number * 100 div Max;
    end;
    HourLabel.Caption := IntToStr (NPrimes) +
      ' below ' + IntToStr (Max);
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

Valahányszor a ciklusmag lefut, a folyamatsáv értéke frissítésre kerül. Feltehetőleg ez az utasítás lassítja le az algoritmust, különösen a vezérlő újrarajzolása miatt. A rutin végén az eredmény a jobb oldali címkébe íródik. Ennek a változatnak a kódja legfőképpen azért szerepel, hogy szembetűnőbb legyen más verzióktól való eltérése. Hibája, hogy a gomb lenyomása után az algoritmus végeztéig az alkalmazás nem reagál az üzenetekre. A Windows 3.1 működésével ellentétben Windows 95 alatt a program futása közben lehet más alkalmazásokkal dolgozni. Ha azonban a BackProc formját egy másik ablak mögé rejtjük, majd újra előtérbe helyezzük, az algoritmus végeztéig – míg az alkalmazás nem dolgozhatja fel az üzeneteket – a form nem lesz újrarajzolva.

Háttértevékenység időzítőkkel

A BackProc példa Timer gombja elindítja a háttértevékenység legrosszabb formáját. Az időzítők használata más megoldásokkal összehasonlítva nagyon lassú kódot eredményez. A `for` ciklus szerepét az időzítő veszi át, egy-egy időzítő esemény alkalmával egyszer lefut a ciklusmag. A gomb OnClick eseményét kezelő eljárás egyszerűen beállítja néhány privát változó kezdeti értékét (ezeket a változókat a formra kellett helyezni, mert két különböző eljárás használja). Miután ez megvan, az időzítő elindul (ez tervezés közben még nem aktív):

```
procedure TForm1.TimerButtonClick(Sender: TObject);
begin
  TimerNPrimes := 0;
  TimerMax := SpinEdit1.Value;
  TimerNumber := 2;
  Timer1.Enabled := True;
  ProgressBar2.Position := 0;
  TimerLabel.Caption := 'Computing...';
end;
```

A tényleges kód a Timer1Timer eljárásban szerepel. A `for` ciklus helyett egy sima `if` utasítás áll, és egy hívás az Inc eljárásra:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if TimerNumber < TimerMax then
  begin
    if IsPrime (TimerNumber) then
      Inc (TimerNPrimes);
    Inc (TimerNumber);
  end
  else
  begin
    Timer1.Enabled := False;
    TimerLabel.Caption := IntToStr (TimerNPrimes) +
      ' below ' + IntToStr (TimerMax);
```

```
        Beep;
      end;
      ProgressBar2.Position :=TimerNumber * 100 div
        TimerMax;
    end;
```

Az így nyert kódot összehasonlíthatjuk az eredeti, `for` cikluson alapuló kóddal. A kód maradék része könnyen megérthető. A kurzor természetesen nem változik, és a folyamat végét egy egyszerű csipogás jelzi.

Vegyük észre, hogy a program működése közben a Timer gomb nincs kikapcsolva, így a felhasználó újraindíthatja az algoritmust. Ezenkívül rákattinthat a Hourglass gombra is, elindítva az algoritmus első változatát, amely átmenetileg megállítja az időzítőn alapuló kódot (az alkalmazás bármely többi kódjához hasonlóan).

Üzenetek feldolgozása a háttérben

Egy sokkal egyszerűbb és jobb módszer az, hogy míg a prímszámokat számolgatjuk az alkalmazás (és Windows 3.1 alatt az egész rendszer) a háttérben feldolgozhatja az érkező üzeneteket. Azért sokkal egyszerűbb ez, mert az algoritmus eredeti változatához csak egyetlen utasítást kell hozzáírni: az Application objektum ProcessMessage eljárásának meghívását.

Az utasítást a `for` ciklus magjába írtuk, így az alkalmazás más eseményei minden egyes lépésnél lekezelhetők:

```
for Number := 2 to Max do
begin
  if IsPrime (Number) then
    Inc (NPrimes);
  ProgressBar3.Position := Number * 100 div Max;
  Application.ProcessMessages;
end;
```

A futást ez folyamatossá teszi. A formon bármit elvégezhetünk minden probléma nélkül – csak sokkal lassabban. Az eddigi kódot összegezve, az Hourglass gombra kattintva a folyamat befejeztéig az alkalmazáson belül minden leáll. A Timer verzió hatására a számolás sokkal lassabb lesz ugyan, de az alkalmazás reagál más üzenetekre. A magas prioritású ProcessMessage verzió gyorsabban végez, és az alkalmazást szintén nem állítja le. Vegyük észre emellett, hogy a ProcessMessage változat újraindítható, még mielőtt befejezné működését (az első számolás megvárja, míg az újabb véget ér).

Ajánlott a példaprogram lefuttatása és az összes módszer kipróbálása. Segíthet a Windows 3.1-stílusú és a párhuzamos programozás közötti különbségek megértésében. Mellesleg számolás közben a Timer verzió is újraindítható, de megsemmisíti az előző eredményét, és újrakezdi a számolást. A

ProcessMessage változat másrészről elindít egy második számoló ciklust, és az eredeti újraindulása problémákat okozhat. A legnagyobb probléma az, hogy valahányszor a függvény elindul, a verem egy jókora része lefoglalásra kerül. Különböző számolások egyidejű futtatása elhasználhatja a program teljes vermét, amit a Windows nagyon nem szeret (a rendszer teljesen leállhat).

Általános célú programozási technika, hogy a ProcessMessage-eket használó kódnak az efféle veremproblémák elkerülése végett védenie kell magát az újbóli indítástól. A legjobb megoldás egy állapotjelző párbeszédablak megjelenítése úgy, hogy a fő alkalmazás-ablak (minden menüjével együtt) takarva legyen. A példaprogramban egyszerűen kikapcsoljuk a megfelelő gombot, így az algoritmus más változatai emellett ugyanúgy futtathatóak.

Háttérszámolás

A BackProc példa alsó gombja által nyújtott megoldás az egyedüli, amelyik a rendszer szabadidejének igénybevételével valódi háttértevékenységet valósít meg. A Timer változathoz hasonlóan az OnIdle eljárás is két részre bontja a kódot. Az OnIdle gombon történő kattintás hatására egy inicializáló rész hajtódik végre, beleértve az Application objektum OnIdle eseménykezelőjének beállítását is:

```
procedure TForm1.IdleButtonClick(Sender: TObject);
begin
  IdleNPrimes := 0;
  IdleMax := SpinEdit1.Value;
  IdleNumber := 2;
  Application.OnIdle := IdleProc;
  ProgressBar4.Position := 0;
  IdleLabel.Caption := 'Computing...';
end;
```

Ez a kód szintén használja a formon definiált változókat (privát mezőket), és `for` ciklus helyett szintén egy ellenőrzésből és növelésből áll. Íme az IdleProc eljárás kódja:

```
procedure TForm1.IdleProc (Sender: TObject;
  var Done: Boolean);
begin
  if IdleNumber < IdleMax then
  begin
    if IsPrime (IdleNumber) then
      Inc (IdleNPrimes);
    Inc (IdleNumber);
    Done := False;
  end
  else
  begin
```

```
        IdleLabel.Caption := IntToStr (IdleNPrimes) +
          ' below ' + IntToStr (IdleMax);
        Done := True;
        Beep;
        Application.OnIdle := nil;
      end;
      ProgressBar4.Position :=IdleNumber * 100 div IdleMax;
    end;
```

A második, Done paraméter jelzi a rendszernek, hogy van-e szükség további háttérszámításra, vagy az algoritmus befejeződött. Az algoritmus befejeztével a nem kívánt további tétlenségi üzenetek elkerülése érdekében még az eseménykezelőt is ki kell kapcsolni.

Vegyük észre, hogy ebben az esetben a kódunk felett csak nagyon korlátozott felügyeletünk van. Nem tudjuk, hogy a program meddig tart. Ha a számítógépen semmi más nem történik, ez a változat ugyanolyan gyors, mint az eredeti; de más műveletek végzésekor, mint például a ProcessMessage eljárás hívásain alapuló verzió futtatásakor a feldolgozás majdhogynem meg is állhat.

A technikáknak nincs egyértelmű nyertese. Valódi háttérszámításhoz az OnIdle a megfelelő választás. Ahhoz, hogy az alkalmazás még a program kemény munkája közben is reagáljon más eseményekre, inkább a ProcessMessage ajánlott. Az időzítők használata a kettő hasznának egy közbenső változata. Ezzel szemben egy valódi párhuzamosan programozott alkalmazás létrehozásához a Win32 platformok szál lehetőségeit és a Delphi ehhez kapcsolódó megfelelő támogatását kell használni.

Párhuzamos programozás a Delphiben

A Windows 95 és a Windows NT lehetőséget ad két eljárás egyidejű futtatására, és a program vezérelheti is ezeket. Mielőtt rátérnénk a párhuzamos programozás megvalósítására, vizsgáljuk meg, miért is van szükség egy adott programban különböző szálak használatára. Mindenekelőtt vegyük figyelembe a párhuzamosság hátrányait:

- A párhuzamosság a program futását lassítja, kivéve, ha a gépben több processzor van, és az operációs rendszer el tudja osztani a szálakat a processzorok között.
- A párhuzamos programoknak összehangoltan kell a megosztott erőforrásokat és a memóriát elérniük, és ez, mint sok esetben látni fogjuk, az írás műveletét sokkal bonyolultabbá teszi.

Szerencsére azért van néhány előnye is ennek a módszernek. Futtathatunk egy szálat a háttérben, miközben a felhasználó tovább használhatja a programot. Prioritásának állításával egy szál futását gyorsabbá tehetjük a többinél, szabályozhatjuk a különböző szálak erőforrás-elérését, stb.

A TThread osztály

Windows API függvényeinek egy csoportja kifejezetten a szálak kezelésére szolgál (a CreateThread kulcsfontosságú), de mivel a Delphi `TThread` osztálya maradéktalanul kiszolgálja igényeinket, azokat nem részletezzük.

Az első dolog, amit a `TThread` osztályról tudnunk kell az, hogy sohasem fogjuk közvetlenül használni, mivel ez egy absztrakt osztály, ami egy virtuális absztrakt eljárással rendelkezik. A szálak használatakor mindig egy alosztályt képzünk a `TThread`-ből, mely tartalmazza az alaposztály minden tulajdonságát. A `TThread` osztály konstruktorának egyetlen paraméterével azt állíthatjuk be, hogy a szálat azonnal, vagy egy kis várakozással indítjuk-e el:

```
constructor Create(CreateSuspended: Boolean);
```

A szálak szinkronizálására a következő eljárások szolgálnak:

```
procedure Resume;
procedure Suspend;
function Terminate: Integer;
function WaitFor: Integer;
```

A tulajdonságok között szerepel a Priority (prioritás), a Suspended (felfüggesztett) és két csak olvasható alacsonyszintű érték (a Handle és a ThreadID). Az osztály emellett rendelkezik egy védett felülettel, mely két eljárást tartalmaz a szálak alosztályaihoz:

```
procedure Execute; virtual; abstract;
procedure Synchronize(Method: TThreadMethod);
```

A virtuális absztrakt eljárásként deklarált Execute metódust minden egyes szál osztálynál újra kell definiálnunk. Ez a szál fő kódját tartalmazza, melyet a Windows API függvényeinek használatakor egy speciális szál függvényben kell elhelyeznünk. A Synchronize eljárás segítségével elkerülhetjük a VCL összetevők egyidejű használatából adódó konfliktusokat. A VCL kód a program fő szálában fut, és az újrabelépési problémák (vagyis előző hívás befejezte előtt a függvénybe történő újabb belépésből eredő problémák) elkerülése végett össze kell hangolni a VCL-hez való hozzáférést. A Synchronize egyetlen paramétere egy paraméterekkel nem rendelkező eljárás, rendszerint ugyanabból a szál osztályból.

A Delphi 2 alkalmazásaiban egy háttérszálnak minden olyan alkalommal, amikor a felhasználói felületet, vagy más VCL komponenst használta, összehangoló hívásokat kellet végeznie. A Delphi 3 már tartalmaz párhuzamos programozást támogató beépített elemeket. A `TCanvas` osztályban lévő Lock és UnLock zároló eljárásokkal például egy háttérszál közvetlenül rajzolhat a főformra. A `TThreadList` egy másik hasznos osztály, mellyel több szál biztonságosan és egyidejűleg férhet hozzá ugyanahhoz a `TList`-hez. A szálak támogatottsága még grafikus elemeknél, akár bitképeknél is megtalálható.

Az első példa

Első egyszerű példának lássunk egy a Synchronize eljárást használó programot, mely egy szál segítségével rajzol a form felületére (a példa neve ThOld). A `TPainterThread` szálosztály felüldefiniálja az Execute metódust és egy saját Paint eljárást definiál. A Paint metódus feladata a VCL objektumokhoz való hozzáférés, így azt csak a Synchronize eljárásból hívhatjuk meg. Mivel a Paint közvetlenül nem fogadhat paramétereket, és a Synchronize-zal kompatíbilisnek kell maradnia, az osztálynak privát adatokat kell tartalmaznia. Íme a szál osztály deklarációja:

```
type
  TPainterThread = class(TThread)
  private
    X, Y: Integer;
  protected
    procedure Execute; override;
    procedure Paint;
end;
```

A Paint metódus a form képpontjait pirosra festi:

```
procedure TPainterThread.Paint;
begin
  Form1.Canvas.Pixels [X, Y] := clRed;
end;
```

A kód nagyon egyszerű, és mivel csak akkor kerül végrehajtásra, amikor a Delphiben nem fut semmi más, az összehangoló eljárások a lehető leggyorsabbak. A szál Execute fő függvénye a Paint függvényt a Synchronize eljárásnak paraméterként átadva véletlenszerűen jelöl meg egy-egy képpontot:

```
procedure TPainterThread.Execute;
begin
  Randomize;
  repeat
    X := Random (300);
    Y := Random (Form1.ClientHeight);
    Synchronize (Paint);
  until Terminated;
end;
```

A 25.10. ábra a program futásának eredményét mutatja. Amint a fenti listában láthatjuk, a szál addig fut, amíg le nem állítják. A főformon egy gomb szerepel a szál elindítására. Ha a Create konstruktornak False-t adunk át, a szál azonnal elindul:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Enabled := False;
```

```
    Button2.Enabled := True;
    PT := TPainterThread.Create (False);   // start
end;
```

A PT a form osztály egy privát `TPainterThread` mezője. Az első gomb elkészíti a szálat, majd elindítja, a második felszabadítja a szál objektumot, és átkapcsolja a gombok Enabled tulajdonságát.

25.10. ábra *A ThOld példa: a színes képpontokat egy háttérben futó szál festi be*

A főform emellett az egérkattintásokat is kezeli; a formra történő kattintáskor a kattintás helye körül kör alakban a vörös képpontok eltűnnek, és felveszik a form színét:

```
procedure TForm1.FormMouseDown(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState;
    X, Y: Integer);
begin
    Canvas.Pen.Color := Color; // of the form
    Canvas.Brush.Color := Color;
    Canvas.Ellipse (x - 30, y - 30, x + 30, y + 30);
end;
```

Egy zároló példa

A Delphi 3 lehetőségeivel az előző példa újraírható. Ez esetben a form Canvasához való hozzáférést annak zárolásával érhetjük el, és a Synchronize hívás el-

kerülésével a kód egyszerűsödik. A ThLock példa szál osztálya csak egy eljárással rendelkezik:

```
type
  TPainterThread = class(TThread)
  protected
    procedure Execute; override;
  end;
```

Az Execute kódja most elvégez mindent, a form Canvas komponensének zárolása utáni kimenetet is beleértve:

```
procedure TPainterThread.Execute;
var
  X, Y: Integer;
begin
  Randomize;
  repeat
    X := Random (300);
    Y := Random (Form1.ClientHeight);
    with Form1.Canvas do
    begin
      Lock;
      try
        Pixels [X, Y] := clRed;
      finally
        Unlock;
      end;
    end;
  until Terminated;
end;
```

Ha ezután a Canvas-t egy OnPaint-től (mely a zárolást automatikusan kezeli) különböző eseménykezelőből szeretnénk elérni, a Lock eljárást újra meg kell hívni:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  Canvas.Lock;
  try
    Canvas.Pen.Color := clYellow;
    Canvas.Brush.Color := clYellow;
    Canvas.Ellipse (x - 30, y - 30, x + 30, y + 30);
  finally
    Canvas.Unlock;
  end;
end;
```

Szinkronizálási lehetőségek

A Delphi 3-ban a Lock eljárás használata az ezzel a lehetőséggel rendelkező néhány objektumra korlátozódik. Mikor különböző szálaknak egyidejűleg kell hozzáférniük ugyanazon komponens tulajdonságaihoz, továbbra is lehetőségünk van a `TThread` osztály védett Synchronize eljárásának használatára. A másodlagos szálak általában olyan háttérműveletek elvégzésére alkalmasak, mint fájlok áthelyezése, vagy bonyolultabb számítások esetében, melyek nem, vagy csak kis mértékben igénylik a felhasználói felület frissítését. Ha a felhasználói felület frissítésére csak korlátozott mértékben van szükség, az előbb említett két technika különböző megközelítései jöhetnek szóba.

Egy szál nyugodtan frissíthet egy adatszerkezetet, melyet a fő szál néha-néha megvizsgál, csak az írás/olvasást kell elkerülnie. Másik lehetőségként használhatjuk a hagyományos Windows-alapú párhuzamos programozást; a szál egy frissítést kérő üzenetet küld a fő ablaknak. Ügyeljünk arra, hogy erre a célra a SendMessage-et (ami egy szinkron hívás, egy közvetlen függvényhíváshoz hasonlít) nem, hanem csak a PostMessage-et (ami egy aszinkron hívás, és az üzenetsort használja) ajánlott használni. Az első kettőhöz képest ezeket a megoldásokat csak ritkán használják, ezért a könyvben sem szerepel rájuk példa.

A szálak prioritása

A szálak harmadik példája az előzőnek egy kiegészítése, de ezúttal az egyes szálak egyidejűleg futnak, és a felhasználó vezérlősávok segítségével módosíthatja a prioritásukat. A `TPainterThread` osztály új változata:

```
type
  TPainterThread = class(TThread)
  private
    Color: Integer;
  protected
    procedure Execute; override;
  public
    constructor Create (Col: TColor);
  end;
```

Az osztály egy konstruktorral bővült, mely egy kezdeti színértéket ad át a szálnak. Ha a Color a szál egy nyilvános mezője, a program közvetlenül tudja módosítani. A konstruktor forráskódja:

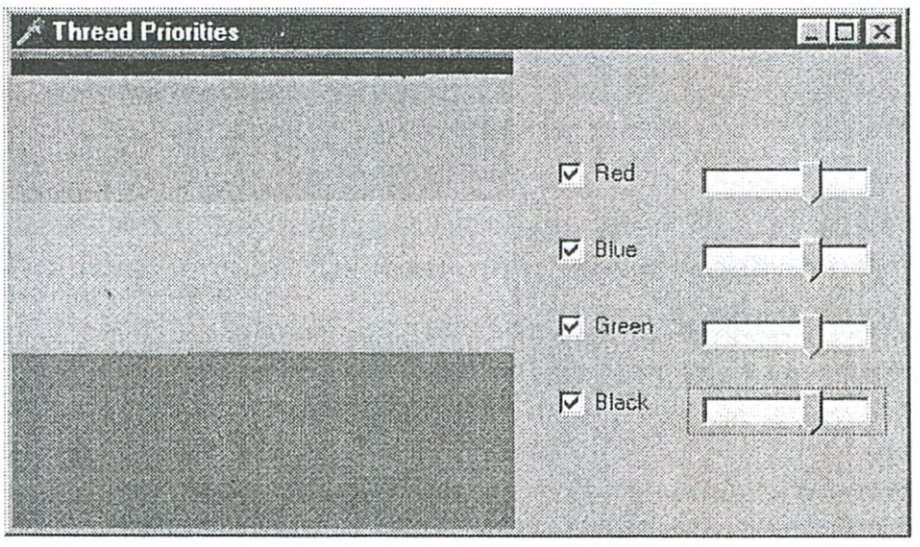
```
constructor TPainterThread.Create(Col: TColor);
begin
  Color:= Col;
  inherited Create (True);
end;
```

A konstruktor inicializálja a privát adatot, majd meghívja az alaposztály konstruktorát, így a szál felfüggesztett állapotba kerül. A szál Execute eljárása egyszerűen végigmegy a képernyő sorain, és minden képpontot egy adott színre fest:

```
procedure TPainterThread.Execute;
var
  X, Y, X1: Integer;
begin
  X := 0;
  Y := 0;
  repeat
    // scan the lines...
    X1 := X + 1;
    X := X1 mod 250;
    Y := Y + X1 div 250;
    Y := Y mod Form1.ClientHeight;
    Form1.Canvas.Lock;
    try
      Form1.Canvas.Pixels [X, Y] := Color;
    finally
      Form1.Canvas.UnLock;
    end;
  until Terminated;
end;
```

A főformon négy jelölőnégyzet és négy vezérlősáv található (ahogyan a 25.11. ábra is mutatja). A form lokális adatokkal is rendelkezik, egy tömbbel a négy szál objektum tárolására:

```
private
  PT: array [1..4] of TPainterThread;public
```



25.11. ábra *A ThPrior példa kimenete. A felhasználói felületet négy szál frissíti párhuzamosan*

A tömb inicializálása a form elkészülésekor történik:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  PT [1] := TPainterThread.Create (clRed);
  PT [2] := TPainterThread.Create (clBlue);
  PT [3] := TPainterThread.Create (clGreen);
  PT [4] := TPainterThread.Create (ClBlack);
end;
```

Vegyük észre, hogy a program által készített négy szál felfüggesztett állapotban van. Csak a megfelelő jelölőnégyzet bejelölésekor indul el, és annak törlésekor a megfelelő szál újra felfüggesztett állásba kerül:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if (Sender as TCheckbox).Checked then
    PT [(Sender as TCheckbox).Tag].Resume
  else
    PT [(Sender as TCheckbox).Tag].Suspend;
end;
```

Ugyanazon eseménykezelő használata érdekében az egyes jelölőnégyzetek Tag tulajdonsága a neki megfelelő szál sorszámát tartalmazza. A folyamatsávoknál, melyek a szálak aktuális prioritásának beállítására szolgálnak, a technika ugyanez:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  PT [(Sender as TTrackBar).Tag].Priority :=
    TThreadPriority ((Sender as TTrackBar).Position);
end;
```

A prioritás beállításakor a vezérlősáv aktuális Position értékét egyszerűen a megfelelő TThreadPriority értékre alakítjuk át. Az így keletkező értéket a Tag tulajdonság által meghatározott megfelelő szál prioritásának beállítására használjuk. A programból sokat lehet tanulni, mert a szál prioritásának módosítása közben a képernyőn azonnal láthatjuk annak hatását.

Szálak szinkronizálása

Korábban láthattuk, hogy a szál és az alkalmazás többi része közötti összhang megteremtésének két megközelítési módja lehetséges: a szál objektum Synchronize eljárásának használatával, és egy olyan VCL osztály Lock metódusának segítségével, amely támogatja azt, mint például a `TCanvas`. Utolsó példánkban a főform Canvas-ának Lock eljárását használva négy olyan szálat hangoltunk össze, melyek a képernyőt festették. Ez azonban egy nagyon speciális eset volt. Két szál összehangolására általában a Windows API-ban rendelkezésre álló alacsonyszintű megoldásokhoz hasonló technikák szükségesek.

A következő bekezdésekben egy egyszerű esetről lesz szó, amelyben egy szál befejezésére várunk. Az ezt követő szakaszokban ennél bonyolultabb példákkal is találkozunk majd.

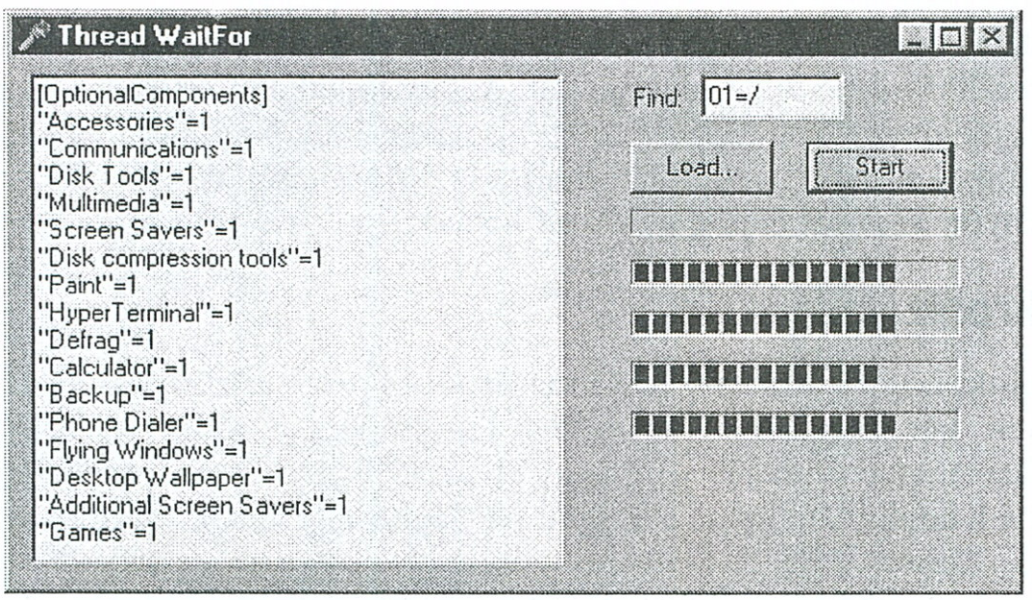
Várakozás egy szálra

Ha egy szálnak meg kell várnia egy másik szál befejezését, egyszerűen meghívhatja a kívánt szálhoz tartozó WaitFor eljárást. Íme egy példának egy részlete, melyben a program elindít egy szálat, majd megvárja, amíg véget ér:

```
Comp := TMyThread.Create (True);
// initialize the thread...
Comp.Resume;
Comp.WaitFor;
// look for final values...
Comp.Free;
```

A kódot nagyon könnyű elkészíteni, ne feledjük azonban, hogy ezt a fő szál részeként nem írhatjuk meg (például egy egyszerű üzenetkezelő függvényben kell ezt megtennünk), ha azzal egy másik szálnak összhangban kellene lennie. Ha a főformhoz kapcsolódó szálban egy másik szál befejezését várjuk, és egy ezektől független szál a felhasználói felülethez szeretne hozzáférni (ennélfogva meg kell várnia, hogy a fő szál elvégezze a műveletét), a program holtpontba jut!

A probléma elkerülése érdekében két szál összehangolására használhatjuk a WaitFor-t. Az első szál készít egy másodikat, és a főform szálának befolyásolása nélkül megvárja, míg az véget ér.



25.12. ábra *A ThWait példában minden szál egy folyamatsávon jelzi az állapotát*

Több szál összehangolásának bemutatására szolgál a következő karakterszámláló program, a ThWait. A program megszámolja, hogy a Memo összetevő szövegében (a szöveget tulajdonképpen egy tetszőleges fájlból is betölthetjük, mindaddig míg a számolást el nem indítjuk) hányszor szerepel a szerkesztőmezőben magadott négy karakter. A program az egyes karaktereket egyszerre keresi a főform által indított több szál segítségével. A 25.12. ábrához hasonlóan a szálak egy folyamatsáv alkalmazásával folyamatosan kijelzik aktuális állapotukat – azaz, hogy a szövegben meddig jutottak el.

A program tényleges „motorja" a `TFindThread` osztály, amely a keresendő karaktert a LookFor mezőjében tárolja, és a folyamatsáv értékére és az állapot frissítésére a Progress mezőt használja. A számlálás eredménye a Found nyilvános mezőben keletkezik:

```
type
  TFindThread = class(TThread)
  protected
    Progr: Integer;
    procedure UpdateProgress;
    procedure Execute; override;
  public
    Found: Integer;
    LookFor: Char;
    Progress: TProgressBar;
  end;
```

A szál magja, ahogy azt megszoktuk, most is az Execute eljárásban található, mely egy adott karaktert keresve végignézi a feljegyzés sorait. Vegyük észre, hogy a feljegyzés tulajdonságaihoz szabadon hozzáférhetünk összehangolás nélkül is, mivel ez a művelet egy nem destruktív olvasás – nincs hatással a Memo komponens állapotára:

```
procedure TFindThread.Execute;
var
  I, J: Integer;
  Line: string;
begin
  Found := 0;
  with Form1.Memo1 do
    for I := 0 to Lines.Count - 1 do
    begin
      Line := Lines [I];
      for J := 1 to Length (Line) do
        if Line [J] = LookFor then
          Inc (Found);
      Progr := I + 1;
      Synchronize (UpdateProgress);
    end;
end;
```

Az UpdateProgress eljárás a Progr mező segítségével egyszerűen frissíti a folyamatsáv állapotát:

```
procedure TFindThread.UpdateProgress;
begin
  Progress.Position := Progr;
end;
```

A szál négy másolatát az elsődleges szál, a TMultiFind osztály aktiválja. Íme ennek az osztálynak a deklarációja:

```
type
  TMultiFind = class(TThread)
  protected
    Progr: Integer;
    procedure UpdateProgress;
    procedure Execute; override;
    procedure Show;
  public
    LookFor, Output: String;
    Progresses: array [1..5] of TProgressBar;
  end;
```

Ez a szál osztály a négy TFindThread objektumot használva megkeresi a LookFor string (a program megfelelő működése érdekében ennek négy karakterből kell állnia) karaktereit:

```
procedure TMultiFind.Execute;
var
  Finders: array [1..4] of TFindThread;
  I: Integer;

begin
  // set up the four threads
  for I := 1 to 4 do
  begin
    Finders[I] := TFindThread.Create (True);
    Finders[I].LookFor := LookFor[I];
    Finders[I].Progress := Progresses [I+1];
    Finders[I].Resume;
  end;

  // wait for the threads to end...
  for I := 1 to 4 do
  begin
    Finders[I].WaitFor;
    Progr := I;
    Synchronize (UpdateProgress);
  end;
```

```
    // show the result
    Output := 'Found: ';
    for I := 1 to 4 do
      Output := Output + Format ('%d %s, ',
        [Finders[I].Found, LookFor[I]]);
    Synchronize (Show);

    // delete threads
    for I := 1 to 4 do
      Finders[I].Free;
end;
```

A `for` ciklust a WaitFor hívással jól figyeljük meg. A végén az Execute eljárás az eredményt a Show összehangolt metódusban jeleníti meg. A szálakat kipróbáló program meglehetősen egyszerű. Ahogyan a 25.12. ábrán láthattuk, tartalmaz egy feljegyzést, ahová egy fájlt tölthetünk, és egy szerkesztőmezőt a négy karakterrel. A karakterek számának ellenőrzése a szerkesztőmező elhagyásakor történik:

```
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  if Length (Edit1.Text) <> 4 then
  begin
    Edit1.SetFocus;
    ShowMessage ('The edit box requires
        four characters');
  end;
end;
```

A Start gomb elindítja a szálat, mely egymás után azonnal aktiválja a másodlagos szálakat:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  if Assigned (MainThread) then MainThread.Free;
  MainThread := TMultiFind.Create (True);
  MainThread.Progresses [1] := ProgressBar1;
  MainThread.Progresses [2] := ProgressBar2;
  ...
  MainThread.Progresses [1].Max := 4;
  for I := 2 to 5 do
    MainThread.Progresses[I].Max := Memo1.Lines.Count;
  for I := 1 to 5 do
    MainThread.Progresses[I].Position := 0;
  MainThread.LookFor := Edit1.Text;
  MainThread.Resume;
end;
```

Vegyük észre, hogy az eljárás végén nem törölhetjük a szálat, mert a WaitFor hívás miatt a program megrekedne. Párhuzamosan programozott alkalmazások készítésekor ügyeljünk az ilyen csapdákra, mert az egész rendszert lefagyaszthatja, és ilyenkor az egyetlen gyógyír a reset gomb használata.

Az operációs rendszer stabilitásának megőrzése érdekében ugyanakkor egy újabb szál létrehozása (mint a fenti kód elején) vagy a program vége előtt nem szabad elfelejtenünk a szál törlését. A szál objektumot ugyanezen okból kell a form privát mezőjeként deklarálnunk, ahelyett, hogy az őt elindító eljárás lokális változója lenne.

Windows összehangolási technikák

A Windows API függvényei számos további összehangolási technikát szolgáltatnak (csak Win32 platformokon):

- A kritikus szakaszok (critical section) a forráskód azon részei, melyeket két szál egyidejűleg nem futtathat. Használatával biztosíthatjuk, hogy a forráskód meghatározott részei egymástól függetlenül hajtódjanak végre. A kritikus szakaszokat csak egy folyamatban, egy alkalmazáson belül alkalmazhatjuk.
- A mutexek segítségével egy bizonyos erőforráshoz való hozzáféréseket különíthetjük el. A mutexet előbb le kell foglalni, elérni az erőforrást, majd felengedni a mutexet. A mutex foglalt állapota alatt egy másik szál (vagy folyamat) azt egészen addig képtelen lefoglalnia mutexet, míg az előző alkalmazás (vagy folyamat) fel nem engedi. Mint a OneCopy4 példában láthattuk, a mutex megosztható különböző alkalmazások között.
- A szemafor a mutexhez hasonló, de számlálási lehetőséget biztosít: megengedhetjük például, hogy egy adott erőforráshoz egyidejűleg legfeljebb hárman férjenek hozzá. A mutex egyenértékű az olyan szemaforral, melynek maximális szálszáma 1.
- A szálakat rendszereseményekkel (mint például a felhasználói fájlműveletek) segítségével is összehangolhatjuk. A Delphi TThread osztályának WaitFor eljárása egy eseményt használ. Emellett az események még több szál egyidejű felébresztésére is használhatók.

Egy példa elkészítése

Mindezen technikák bemutatása céljából készült a ThSynch példa. Tegyük fel, hogy két szál egy karakterláncon végez műveleteket, a string értékét használva és frissítve azt. Emellett tegyük fel, hogy a string értéke a két szál között megosztott. A példában a karakterlánc kezdetben 20 darab A karaktert tartalmaz, majd 20 B karakterre frissül, és így tovább. A programban minden szál egyszerűen kiszámolja a string következő értékét, majd saját listapanelébe teszi.

Tipp

A szálaknak természetesen nem ajánlatos globális változókat használniuk. A Delphi ez irányban a szálak osztályokba sorolásával és a `threadvar` kulcsszó használatával szál változók definiálására ad lehetőséget.

A ThSynch példában ténylegesen négy form van, egyenként két listapanellel, egy nem proporcionális Courier betűkészlettel, és a megfelelő szál elindítására használható gombbal. Emellett a főformon négy gomb is szerepel, a hozzá tartozó másodlagos formok részére.

Így végül négy változatot kapunk alapjaiban ugyanazon formból és szál osztályból. Mindegyik ilyen form Start gombja egyszerűen elkészíti egy szál két példányát, és egy-egy listapanelhez rendeli azokat. Egy példa erre:

```
procedure TForm2.BtnStartClick(Sender: TObject);
begin
  ListBox1.Clear;
  ListBox2.Clear;
  Th1 := TListThread.Create (True);
  Th2 := TListThread.Create (True);
  Th1.LBox := Listbox1;
  Th2.LBox := Listbox2;
  Th1.Resume;
  Th2.Resume;
end;
```

A szál osztályok deklarációja a formokat definiáló egységekben szerepel, így nem használunk egyszerre túl sok fájlt. Ugyanezen egység tartalmazza a két szál objektum által használt karakterlánc változót:

```
var
  Form2: TForm2;
  Letters: string = 'AAAAAAAAAAAAAAAAAAAA';
```

Nos, ez messze nem elegáns, de ebben a programban nézzük inkább a végeredményt, és felejtsük el a programozási stílust.

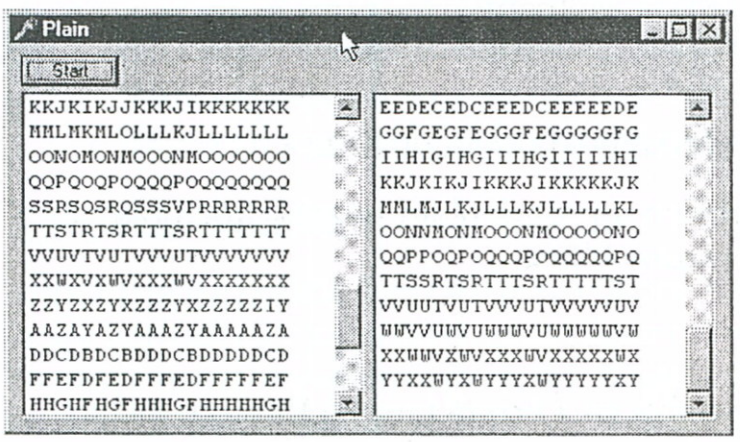
Az egyszerű szál

Íme az első egyszerű példa szál osztálya és a hozzá tartozó Execute eljárás:

```
type
  TListThread = class (TThread)
```

```
    private
      Str: String;
    protected
      procedure AddToList;
      procedure Execute; override;
    public
      LBox: TListBox;
    end;

procedure TListThread.Execute;
var
  I, J, K: Integer;
begin
  for I := 0 to 50 do
  begin
    for J := 1 to 20 do
      for K := 1 to 2601 do // useless repetition...
        if Letters [J] <> 'Z' then
          Letters [J] := Succ (Letters [J])
        else
          Letters [J] := 'A';
    Str := Letters;
    Synchronize (AddToList);
  end;
end;
```

Az AddToList metódus az Str stringet egyszerűen hozzáadja a szálhoz kapcsolódó listapanelhez. Minden számolást direkt hosszabbá tettünk azzal, hogy egyszeri növelés helyett 2601-szer növelünk: A hatás ugyanaz, de több olyan változás van, ahol a két szál konfliktusba kerülhet egymással. Ezt a hatást a 25.13. ábrán láthatjuk. Még jobb, ha sorban két, vagy három Start gombot is lenyomunk, és egyszerre több különböző szálat indítunk el, növelve az esetlegesen felmerülő hibák számát.



25.13. ábra *A ThSynch példa első másodlagos formja hibákat mutat a két listapanel értékei között*

Kritikus szakaszok használata

Két szál műveleteinek elkülönítésére és a szálak tevékenységeinek nagyobb felügyeletére használható a korábban felsorolt Windows összehangoló technikák egyike. Ebben a második változatban a kritikus szakaszokat fogjuk használni. Ehhez a kritikus szakasz részére egy újabb globális változót (vagy a form osztály egy mezőjét) kell deklarálnunk:

```
var
  Critical1: TRTLCriticalSection;
```

Két API hívás segítségével a változó a form elkészülésekor inicializálható és a végén megsemmisíthető:

```
procedure TForm3.FormCreate(Sender: TObject);
begin
  InitializeCriticalSection (Critical1);
end;
procedure TForm3.FormDestroy(Sender: TObject);
begin
  DeleteCriticalSection (Critical1);
end;
```

A kritikus szakaszok használatával a kód bizonyos részeit, mint például a karakterlánc egyes karaktereit frissítő kódokat elkülöníthetjük egymástól. Íme a szál objektum Execute metódusának átalakított változata:

```
procedure TListThread.Execute;
var
  I, J, K: Integer;
begin
  for I := 0 to 50 do
  begin
    EnterCriticalSection (Critical1);
    for J := 1 to 20 do
      ... // same code as above
    Str := Letters;
    LeaveCriticalSection (Critical1);
    Synchronize (AddToList);
  end;
end;
```

A kód hatására amíg az egyik szál az új stringet számolja, addig a másik vár, mielőtt ugyanezt tenné, így a keletkező karakterláncok minden esetben ugyanannak a karakternek 20 példányát tartalmazzák.

Mutex használata

Most ugyanazt a kódot kritikus szakaszok helyett mutex használatával írjuk meg. Az eredmény ugyanaz, de ezt a technikát is szeretnénk bemutatni. Egy-

szerűen csak egy THandle típusú hMutex változót kell deklarálnunk, majd az előző példa négy API hívását az alábbi néggyel kell helyettesítenünk:

```
// in FormCreate
hMutex := CreateMutex (nil, false, nil);
// in FormDestroy
CloseHandle (hMutex);
// in the for loop
WaitForSingleObject (hMutex, INFINITE);
...
ReleaseMutex (hMutex);
```

A TCriticalSection VCL objektum használata

A Delphi 3 kliens/szerver változata egy további lehetőséget nyújt. A SyncObjs unit néhány VCL osztályt definiál az összehangoló objektumok részére: az eseményekre és a kritikus szakaszokra. Felépíthetjük tehát példánk negyedik verzióját, mely nagyon hasonlít a kritikus szakaszon és a Windows API hívásokon alapuló másodikhoz. A különbség az, hogy az objektumot ez esetben így definiáljuk:

```
var
  Critical1: TCriticalSection;
```

és így használjuk:

```
// in FormCreate
Critical1 := TCriticalSection.Create;
// in FormDestroy
Critical1.Free;
// in the for loop
Critical1.Enter;
...
Critical1.Leave;
```

Kevés a különbség, de a kódot kicsivel könnyebb így megírni. A SyncObjs egységben található a TCriticalSection osztály deklarációja a THandleObject, a TEvent és a TSingleEvent osztályéval egyetemben.

Összegezve a hosszú példánkban leírt megoldásokat, a mutex, a kritikus szakasz illetve a szemafor használatára akkor van szükségünk, ha ugyanazon erőforrást, vagy adatot két szál között szeretnénk megosztani. Máskülönben a rendszer szálról-szálra adná át a vezérlést még mielőtt bármelyik is befejezné az aktuális műveletet. Ebben a közbenső állapotban az adat hibás lehet, a többi szál azonban ugyanúgy használná. Az összehangoló objektumokhoz a Windows API hívásaival, vagy a még egyszerűbb VCL objektumokkal férhetünk hozzá.

A Screen objektum használata

Korábban már megvizsgáltuk az Application objektum néhány tulajdonságát és eseményét, vannak azonban más globális objektumok is. Az egyik ilyen a Screen objektum. Segítségével a rendszer megjelenítési módjáról vagy egy futó alkalmazás pillanatnyi formjairól szerezhetünk információt. Az objektum ezenkívül tartalmazza a rendelkezésre álló betűkészletek listáját, melyet már más példákban (mint például a 11. fejezet FontGrid példájában) használtunk.

Képernyőinformációk lekérdezése

Következő példánkban a Screen objektummal végezhető műveletek közül láthatunk néhányat. Először egy formot készítünk néhány címkével és két listapanellel. Az első listapanelben a rendszerbe telepített betűkészleteket, míg a másodikban az alkalmazás pillanatnyi formjainak listáját láthatjuk.

Megjegyzés

A speciális VCL egységek által deklarált más Delphi globális objektumok a Printer és a Clipboard, melyek a 28. illetve a 30. fejezetben kerülnek ismertetésre.

Mindkét listapanelt a Screen objektum tömbtulajdonságainak segítségével töltjük ki. A második listapanel kezelése azonban összetettebb, mert új form létrehozásakor, egy létező megsemmisítésekor, vagy a program aktív formjának megváltozásakor frissítenünk kell. Megjegyezném, hogy a Screen objektum által nyilvántartott formok nem a rendszer, hanem az alkalmazás formjai. Ennek működését megláthatjuk, ha a New címkével ellátott gomb segítségével egy pár új formot készítünk:

```
procedure TMainForm.NewButtonClick(Sender: TObject);
var
  NewForm: TSecondForm;
begin
  {create a new form, set its caption, and run it}
  NewForm := TSecondForm.Create (self);
  Inc (Counter);
  NewForm.Caption := 'Second ' + IntToStr (Counter);
  NewForm.Show;
end;
```

A program egyik kulcsfontosságú részét képezi az OnCreate eseménykezelő, mely a Screen objektum számos jellemzőjét felhasználja:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  {compute screen size}
  ScreenLabel.Caption := ScreenLabel.Caption + ' ' +
    IntToStr(Screen.Width)+'x'+IntToStr(Screen.Height);
  {display fonts and forms data}
  FontsLabel.Caption := 'Fonts: ' +
    IntToStr (Screen.Fonts.Count);
  FontsListBox.Items := Screen.Fonts;
  FillFormsList (self);
  {set the secondary forms counter to 0}
  Counter := 0;
  {set an event handler on the screen object}
  Screen.OnActiveFormChange := FillFormsList;
end;
```

A formlista kezelése

Ahogyan az imént láttuk, az OnCreate eseménykezelő feltölti a listapaneleket, és megadja a képernyő méretét. A Forms listapanelt feltöltő kód egy másik, FillFormsList eljárás belsejében van, melyet a Screen objektum OnActiveFormChange eseményének eseménykezelőjeként telepítünk:

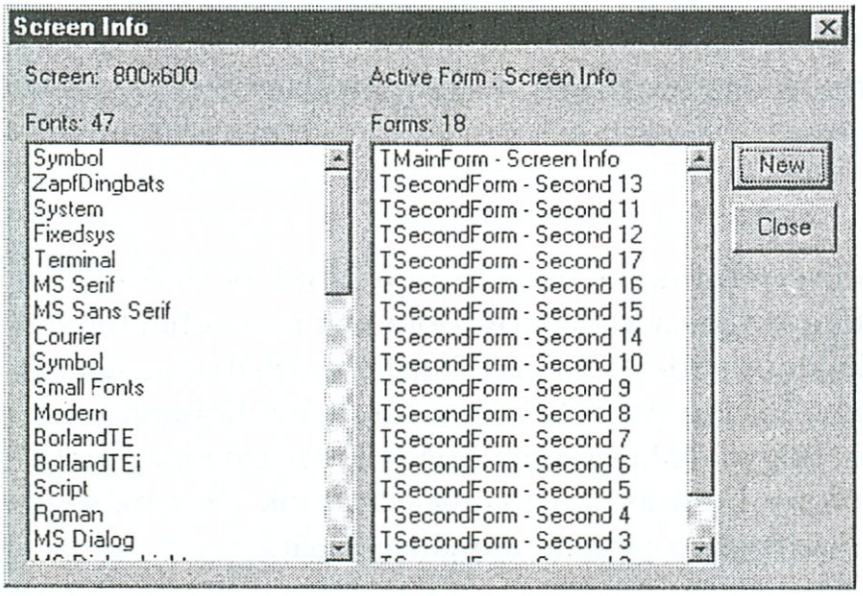
```
procedure TMainForm.FillFormsList (Sender: TObject);
var
  I: Integer;
begin
  FormsLabel.Caption := 'Forms: ' +
    IntToStr (Screen.FormCount);
  FormsListBox.Clear;
  {write class name and form title to the list box}
  for I := 0 to Screen.FormCount - 1 do
    FormsListBox.Items.Add (Screen.Forms[I].ClassName +
      ' - ' + Screen.Forms[I].Caption);
  ActiveLabel.Caption := 'Active Form : ' +
    Screen.ActiveForm.Caption;
end;
```

A FillFormsList eljárás feltölti a listapanelt, és beállítja a felette levő két címke értékét, melyek a formok számát és az aktuális form nevét tartalmazzák. A New gomb hatására elkészül a másik form egy újabb példánya, kap egy új címkét és megjelenik. Az OnActiveFormChange eseményhez telepített kezelő miatt a Forms listapanel frissítése automatikusan megtörténik. A 25.14. ábrán a program futásának eredménye látható, pár új form létrehozása után.

Figyelem!

Az OnActiveFormChange esemény kezelőjét nagyon fontos eltávolítanunk az alkalmazásból való kilépés, azaz a főform megsemmisülése előtt. Máskülönben a kód úgy fut le, hogy nem létezik a listapanel, és egy rendszerhibát kapunk. Ennek elkerülése végett kezeljük le a főform OnClose eseményét, és a `Screen` objektum `OnActiveFormChange` eseményéhez rendeljünk `nil`-t.

Minden egyes újonnan készített form rendelkezik egy Close gombbal, mellyel azok eltávolíthatók. A kód az Action paraméter caFree-re állításával kezeli az OnClose eseményt, így a form bezárásakor az automatikusan megsemmisül. Ez a kód azonban nem működik. A Windows, mielőtt megsemmisít egy formot, a fókuszt átteszi egy új aktív formra, így a lista a bezárt form nevét továbbra is tartalmazni fogja.



25.14. ábra *A Screen példa futás közben, számos másodlagos formmal*

Megjegyzés

Vegyük észre, hogy a program mindig frissíti a második listapanel fölötti ActiveLabel szövegét, mely a listapanel első elemével egyező éppen aktív formot mutatja.

A problémát első ránézésre meg tudnánk oldani, ha a FillFormsList eljárást közvetlenül a másodlagos formból hívnánk meg. Ez azonban azért nem működik, mert a hívás pillanatában a másodlagos form még létezik. Újabb ötlet lehet a wm_ParentNotify Windows üzenet használata, valahányszor egy form elkészül vagy megsemmisül. Ez az üzenet azonban szintén túl korai, mivel még a gyermek ablak ténylegesen nem semmisült meg, és nem törlődött a `Screen.Forms` listából.

A megoldást egy késleltetés bevezetése szolgáltatja, egy felhasználó által definiált Windows üzenet küldése. Mivel a kiküldött üzenet először egy sorba kerül, ha a másodlagos form életének lehető legvégső pillanatában küldjük az üzenetet, a főform már annak megsemmisülése után fogja azt megkapni.

A trükk az üzenetnek a másodlagos form OnDestroy eseménykezelőjéből történő küldése. Ennek megvalósítása érdekében az egység implementációs részébe egy megfelelő `uses` utasítás beírásával a MainForm objektumra kell hivatkoznunk. A wm_User kiküldött üzenetet a főform egy speciális üzenetkezelő eljárása fogadja a következőképpen:

```
public
  procedure ChildClosed (var Message:TMessage);
    message wm_User;
```

Az eljárás kódja a következő:

```
procedure TMainForm.ChildClosed (var Message:TMessage);
begin
  FillFormsList (self);
end;
```

Itt az a baj, hogy ha a másodlagos formokat megelőzően zárjuk be a fő ablakot, akkor a főform még létezik ugyan, de a kódját továbbra már nem lehet futtatni. Egy újabb rendszerhiba (általános védelmi hiba) elkerülése céljából az üzenetet csak akkor küldjük el, ha a főform nincs bezárva. Na de ezt honnan tudjuk meg? Az egyetlen megoldás, hogy a TMainForm osztályt kiegészítjük egy jelzőbittel, melynek értékét a főform bezárásakor megváltoztatunk, és a másodlagos ablak kódjából leellenőrizhetjük a jelzőbit aktuális állását.

Ez egy jó módszer – olyannyira, hogy a VCL-ben is szerepel valami hasonló. Létezik egy alig dokumentált ComponentState tulajdonság. Ez egy Pascal halmaz, mely más jelzőbitek mellett egy csDestroying nevűt is tartalmaz, mely a form bezárásakor állítódik át. Ezek alapján megírhatjuk az alábbi kódot:

```
procedure TSecondForm.FormDestroy(Sender: TObject);
begin
  if not (csDestroying in MainForm.ComponentState) then
    PostMessage (MainForm.Handle, wm_User, 0, 0);
end;
```

A kód hatására a második listapanel mindig tartalmazza az alkalmazás összes formját. Vegyük észre, hogy a másodlagos form automatikus létrehozását a Project Options párbeszédablak **Forms** lapján ki kell kapcsolnunk.

A programhoz utoljára hozzáadott szolgáltatás nagyon egyszerű. A listapanel egy elemére történő kattintáskor a BringToFront metódus használatával a megfelelő form aktiválódik:

```
procedure TMainForm.FormsListBoxClick(Sender: TObject);
begin
  Screen.Forms [FormsListBox.ItemIndex].BringToFront;
end;
```

Ez most már nagyon szép – azaz majdnem szép. Az egyetlen gond az, hogy egy inaktív form listapaneljére kattintva először aktiválódik a főform, és a listapanel újrarendeződik, emiatt végül esetleg a várttól különböző formot választunk ki. A programmal való kísérletezés után hamar rájövünk, hogy ez mit is jelent. Ez a kisebb pontatlanság a programban jó példája annak, amikor a felhasználó olyan információn dolgozhat, amit ugyanakkor dinamikusan frissítünk.

Állapotinformációk elmentése

Ha az alkalmazás állapotáról szeretnénk elmenteni információkat, hogy azokat a program következő indításakor majd előhívjunk az adatot egy fájlban tárolhatjuk olyan formában amilyenben akarjuk. A Windows kifejezetten támogatja az ilyen információkat. A Windows előző verzióiban a szabványos megközelítés egy inicializálós (.INI) állomány készítése volt. Windows 95 (és Windows NT) alatt továbbra is használhatjuk az INI fájlokat, de a Microsoft e helyett a System Registry-t ajánlja.

Windows 95 alatt fontos mindkét technika megismerése, mert a rendszer információk gyakran a Registryben és a jó öreg WIN.INI és SYSTEM.INI állományokban találhatók. Nem áll szándékomban részletezni a két megközelítés előnyeit és hátrányait, de tudnunk kell, hogy léteznek, és a Delphiben könnyen hozzáférhetők. A két technika bemutatása céljából elkészítettem egy program két változatát is, mely elmenti főablakának pozícióját, majd a következő futtatáskor előhívja azt. Az első példa egy INI állományt lesz, míg a második a Registryt használja.

Windows INI fájlok használata

A Delphiben található egy INI állományok kezelésére használható `TIniFile` nevű osztály. Amint egy ilyen osztály egy objektumát elkészítjük és hozzákapcsoljuk egy fájlhoz, információt olvashatunk belőle, és írhatunk bele. Az objektum létrehozásához a konstruktort kell meghívni a megfelelő állomány nevével, a következő kódhoz hasonlóan:

```
var
  IniFile: TIniFile;
begin
  IniFile := TIniFile.Create ('inione.ini');
```

Az INI állomány elhelyezésére két választási lehetőségünk van. A fenti kód beírásával a fájl a Windows könyvtárban fog szerepelni (hacsak nem létezik az alkalmazás könyvtárában egy INIONE.INI állomány). A biztonság kedvéért jobb a `TIniFile.Create` konstruktornak teljes elérési utat átadni. A program első parancssori paraméterének segítségével egyszerűen megkaphatjuk az elérési utat, majd az ExtractFilePath használatával hozzácsatolhatjuk az aktuális fájlnevet:

```
var
  IniFile: TIniFile;
  FileName: string;
begin
  FileName:=ExtractFilePath(ParamStr (0))+'Inione.ini';
  IniFile:=TIniFile.Create(FileName);
```

A másik megoldás az összes INI fájlnak a Windows könyvtárban történő elhelyezése. Ha a feljebbi utasításhoz hasonlóan csak fájlnevet közlünk, ez az alapértelmezés.

Az INI állományok formátuma némi magyarázatra szorul. Ezek a fájlok szakaszokra vannak osztva, mindegyiket egy szögletes zárójelekbe zárt név jelöli. Az egyes szakaszok több elemet tartalmazhatnak a lehetséges három típus közül: karakterláncok, egészek, vagy logikai értékek. Az INI fájlok szerkezetének megismerése érdekében ajánlott valamelyiket megnyitni egy egyszerű szövegszerkesztőben, például a Windows Jegyzettömbben.

Figyelem!

Bár én az alkalmazás könyvtára felőli megközelítést részesítem előnyben, ebben a példában a Windows könyvtár szerepel annak érdekében, hogy a példaprogramot közvetlenül a CD-ről is lehessen futtatni, mivel az egy csak olvasható egység.

A különböző típusú adatok írására a `TIniFile` osztály három eljárását használhatjuk: WriteBool, WriteInteger és WriteString. Más metódusokkal egy egész szakaszt olvashatunk vagy törölhetünk. A Read eljárásokban egy alapértelmezett értéket is megadhatunk arra az esetre, ha az INI állományban a keresett bejegyzés nem található.

IniOne példánk a főform helyének, méretének és állapotának (normál, teljes méretű, vagy kis méretű) elmentésére egy INI fájlt alkalmaz. A példában felmerülő egyetlen probléma az, hogy a VCL-ben az állapot tulajdonság értékének frissítése nem mindig megfelelő, emiatt egy további ellenőrzést kell bevezetnünk annak megerősítésére, hogy a form kis méretű-e.

Az IniOne példa főformja csak egy sima, üres form, egyetlen komponens nélkül. A program két eseményt kezel: Az INI fájl létrehozására vagy megnyitására és a kezdeti értékek beolvasására használt OnCreate-et és a felhasználó jóváhagyása után az állapot mentését végző OnClose-t. Íme az első, a FormClose nevű eljárás kódja, mely a program következő futása esetén visszaolvasásra kerülő adatok mentésére szolgál:

```
procedure TForm1.FormClose(Sender: TObject;
  var Action: TCloseAction);
var
  Status: Integer;
begin
  if MessageDlg ('Save the current
    status of the form?', mtConfirmation,
    [mbYes, mbNo], 0) = IdYes then
  begin
    case WindowState of
      wsNormal: begin
        {save position and size, only if
         the state is normal}
        IniFile.WriteInteger('MainForm','Top',Top);
        IniFile.WriteInteger('MainForm','Left',Left);
        IniFile.WriteInteger('MainForm','Width',Width);
        IniFile.WriteInteger('MainForm',
            'Height',Height);
        Status := 1;
      end;
      wsMinimized: Status := 2;
        {useless: this value is never set by VCL!}
      wsMaximized: Status := 3;
    end;
    {check if the window is minimized, that is,
    if the form is hidden and not active}
    if not Active then
      Status := 2;
    {write status information}
    IniFile.WriteInteger('MainForm', 'Status', Status);
  end;
  {in any case destroy the IniFile object}
  IniFile.Destroy;
end;
```

A kód hátralévő része a FormCreate metódusban szerepel, mely egyszerűen beolvassa a kimentett adatokat és beállítja a korábbi állapotot. Az egyetlen problémát a kis méretű form visszaállítása okozza. A WindowState tulajdonság wsMinimized-re állítása nem működik (a form egy ikonná zsugorítva a Tálcára kerül); az `Application.Minimize` meghívása szintén nem alkalmas a feladat elvégzésére. Egy lehetséges megoldás a formnak röviddel a keletkezése után történő kis méretre állítása Windows API oldalról: egy üzenet küldésével szimuláljuk a felhasználó beavatkozását (a rendszermenü Minimize parancsának kiválasztását). A teljes kód:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Status: Integer;
begin
  IniFile := TIniFile.Create ('ini_one.ini');
  {try to read a value and test if it exists}
  Status :=IniFile.ReadInteger('MainForm', 'Status', 0);
  if Status <> 0 then
  begin
    {read position and size using default values}
    Top :=IniFile.ReadInteger('MainForm','Top',Top);
    Left :=IniFile.ReadInteger('MainForm','Left',Left);
    Width := IniFile.ReadInteger('MainForm',
        'Width', Width);
    Height := IniFile.ReadInteger ('MainForm',
        'Height', Height);
    {set the minimized or maximized status}
    case Status of
      {1: WindowState := wsNormal;
        // this is already the default}
      2: PostMessage (Form1.Handle,
        wm_SysCommand, sc_Minimize, 0);
      3: WindowState := wsMaximized;
    end;
  end;
end;
```

A kódban szerepel egy `TIniFile` típusú IniFile mező, mely a TForm1 osztály privát részéhez tartozik. A programhoz nem tartozik ábra, mivel egy üres form vagy egy ikon nem igazán mondana sokat. Ehelyett ajánlott inkább a programot többször futtatni, átméretezve és áthelyezve ablakát minden alkalommal. Íme a program által készített INI fájlok egy példája:

```
[MainForm]
Top=359
Left=567
Width=217
Height=201
Status=1
```

A Registry használata

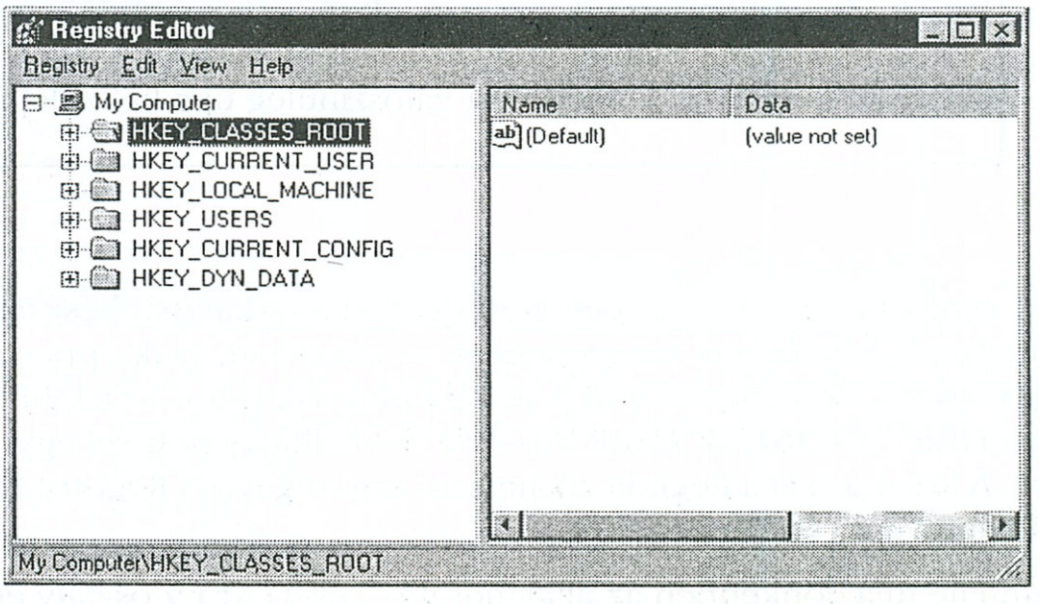
Most már megírhatunk egy hasonló, az INI állományok helyett a rendszer Registry-jét használó programot. Ez előtt azonban vizsgáljuk meg egy kicsit a Registry szerepét és szerkezetét.

Mindent összevetve a Registry a számítógép és a szoftver konfigurációjának és a felhasználói beállításoknak egy hierarchikus adatbázisa. A Registry kezelésére a Windows számos API függvénnyel rendelkezik: lényegében egy kulcsot (vagy egy mappát) kell nyitnunk, majd alkulcsokkal (vagy almappákkal) és értékekkel (vagy elemekkel) dolgozhatunk, de tisztában kell lennünk a Registry szerkezetével és részleteivel. Amint a 25.15. ábra mutatja, a Windows 95 Registry-je hat felsőszintű kulcson alapszik.

A Registry összes részének szerepét nem tárgyaljuk részletesen, a Registry felépítésének részleteit és a saját kulcsok bevezetéséről szóló információkat az ide vonatkozó Microsoft dokumentációban (mint például a Windows 95 vagy

Tipp

Az INI fájlokra vonatkozóan ne feledjük, hogy a Delphi meglehetősen gyakran használja őket, még ha különböző névvel rendelkeznek is. A desktop (.DSK) és options (.DOF) állományok például az INI fájlok szerkezetét követik.



25.15. ábra *A Registry Editor program a Windows 95 Registry hat felsőszintű kulcsát mutatja*

Windows NT Resource Kit) ajánlatos keresni. A Registry jelentőségét nem szabad alábecsülni. A Registry fontos információkat tartalmaz a rendszer hardver konfigurációjáról, a Vezérlőpult beállításairól, az OLE szerverekről, és még a gépre vonatkozó statisztikák is megtalálhatók itt. A Windows 95 alatti RegEdit, vagy a Windows NT alatti RegEdt32 programokkal tanulmányozhatjuk a Registry szerkezetét, és megvizsgálhatjuk a kulcsok aktuális beállításait.

A Delphiben a Registry használatának alapvetően két megközelítése létezik egy-egy VCL osztályon keresztül: a `TRegistry` és a `TRegIniFile`. Az elsőben a Registry API egy általános összefoglalását találhatjuk, míg a másik egy felületet (eljárásokat és tulajdonságokat) biztosít a `TIniFile` osztály részére, az adatokat azonban az állományok használata helyett a Registrybe menti. Utolsó példánk Registry változatának megírásához magától értetődően ezt az osztályt választjuk; használatával a forráskódon nem kell sokat változtatni (valahányszor birtokunkban van egy INI állományokon alapuló kód, ez tiszta nyereség). Az IniOne program három szükséges változtatása:

- Az IniFile objektum osztályaként a `TIniFile` helyett a `TRegIniFile`-t használjuk.
- A FormCreate eljárásban a `TIniFile` objektum helyett egy új `TRegIniFile` objektumot készítünk:

```
IniFile := TRegIniFile.Create ('IniOne.ini');
```

- Az egység `interface` részének `uses` utasításában az IniFiles egységet cseréljük ki a Registry egységgel.

Figyelem!

A RegEdit-et emellett a Registry értékeinek módosítására is használhatjuk, de jobban tesszük, ha csak akkor változtatunk bármit is, ha biztosak vagyunk benne, hogy tudjuk, mit csinálunk. Ha a Registry súlyosan megsérül, a Windowst valószínűleg újra kell telepíteni.

Könnyű, nemdebár? Ezekkel az egyszerű módosításokkal már kész is a Registr példa, melynek lehetőségei megegyeznek az előzőével, csak éppen adatait a Registrybe menti. A `TRegIniFile` osztály használatakor a Delphi tulajdonképpen a HKEY_CURRENT_USER kulcs alá INI fájlkiterjesztéssel egy alkulcsot hoz létre. A kód hatását a RegEdit alkalmazás segítségével a Registry-t megvizsgálva láthatjuk a 25.16. ábra szerint.

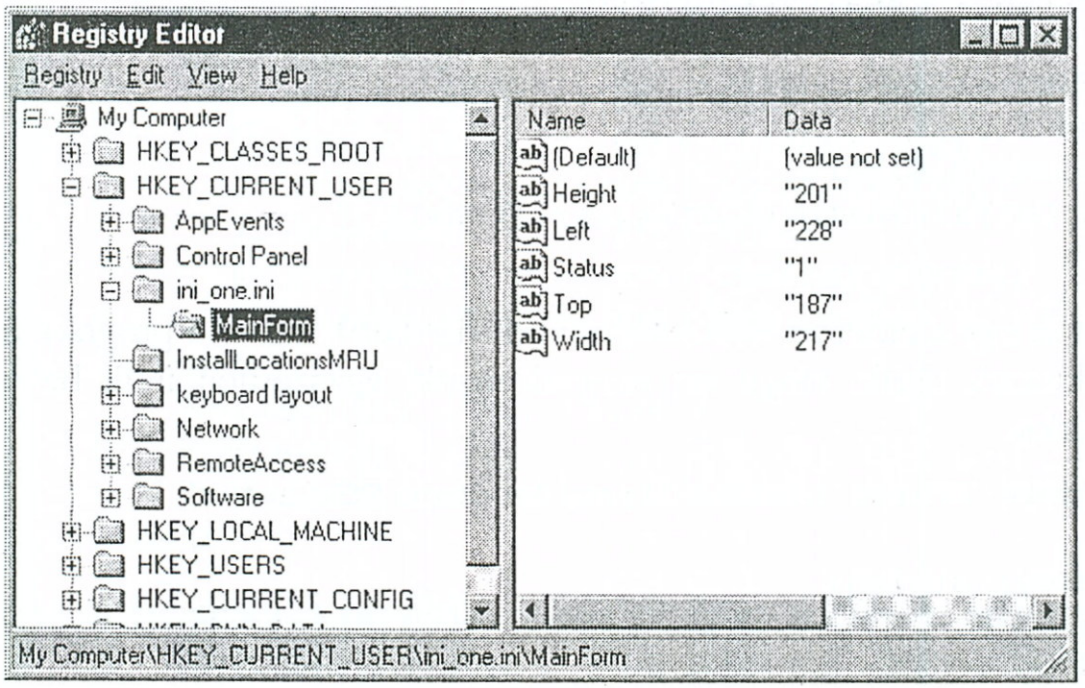
A TRegIniFile tulajdonképpen az általánosabb `TRegistry` osztály egy alosztálya, mely számos a Registry API függvényeihez nagyon hasonló eljárást tar-

talmaz. Ezeknek a függvényeknek a használata nem olyan egyszerű, így a legtöbb esetben maradjunk az egyszerűbb `TRegIniFile`-nál. A Registry használatához először egy kulcsot kell nyitnunk, majd hozzáférhetünk annak adataihoz, beleértve az értékeket és az alkulcsokat.

A `TRegistry` osztály alapvető lehetőségeinek bemutatására szolgál egy nagyon egyszerű, a regisztrációs adatbázist megjelenitő alkalmazás. A program segítségével megnézhetjük a Registry szerkezetét és a kulcsok és elemek értékeit, nem láthatjuk azonban a kulcsokhoz és elemekhez kapcsolt pillanatnyi adatokat. A program a RegEdit alkalmazás lehetőségeinek csak egy részével rendelkezik, de ettől függetlenül egy érdekes példa.

Tipp

A bejegyzéseket ne közvetlenül a HKEY_CURRENT_USER kulcs alá, hanem a SOFTWARE alkulcs alá tegyük, és nem árt a szoftvercéget megnevező újabb szint bevezetése sem, mint például „Software\SoftHouse\Demo".



25.16. ábra *Ahogyan azt a RegEditnél láthattuk, a Registr program használatával a regisztrációs adatbázist új elemekkel bővíthetjük*

A RegView példa egy formon alapszik, mely két kombinált listát és két listapanelt tartalmaz. Az alkalmazás elindulásakor egy TRegistry objektum keletkezik:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Reg := TRegistry.Create;
  Reg.OpenKey ('\', False);
  UpdateAll;
  // select the current root
  ComboKey.ItemIndex := 1;
  ComboLast.Items.Add('\');
  ComboLast.ItemIndex := 0;
end;
```

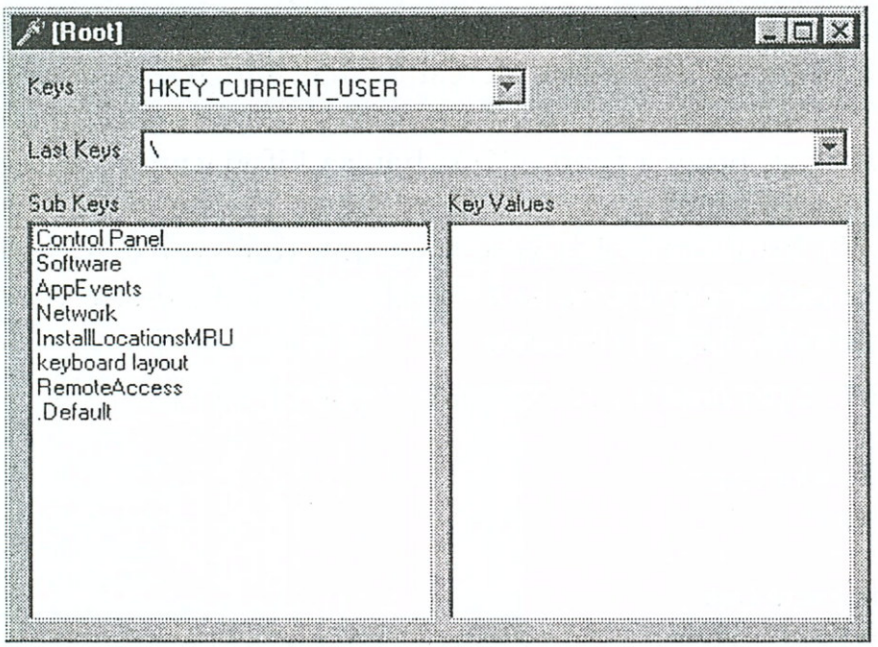
A fenti kód megnyitja a („\" karakterrel jelölt) alapértelmezett gyökérkulcsot, majd frissíti a felhasználói felületet. A végén kiválasztja az első kombinált lista (erről rövidesen bővebben is esik majd szó) alapértelmezett gyökérkulcsát, és az aktuális elemet a ComboLast kombinált listába teszi. Az UpdateAll metódus a pillanatnyi elérési utat bemásolja a form címkéjébe, és a két listapanelt feltölti az alkulcsokkal és az aktuális kulcs értékeivel (amint az a 25.17. ábrán szerepel):

```
procedure TForm1.UpdateAll;
begin
  Caption := Reg.CurrentPath;
  if Reg.HasSubKeys then
    Reg.GetKeyNames(ListSub.Items)
  else
    ListSub.Clear;
  Reg.GetValueNames(ListValues.Items);
end;
```

Ha kiválasztunk egy elemet az első listapanelről (ListSub), akkor a program a megfelelő alkulcsra ugrik. Hogy ezt megvalósítsuk, egyszerűen írjuk be a következő kódot:

```
procedure TForm1.ListSubClick(Sender: TObject);
var
  NewKey: string;
begin
  NewKey := ListSub.Items [ListSub.ItemIndex];
  Reg.OpenKey (NewKey, False);
  UpdateAll;
end;
```

Mindez elég a teljes fa bejárásához. A két kombinált lista némileg bővíti a program lehetőségeit. Az első a Windows 95 lehetséges gyökérkulcsait listázza. Ha a kiválasztás megváltozik, annak tartalma a TRegistry objektum gyökérkulcsaként lesz kijelölve:



25.17. ábra *A RegView példa a Registry kulcsait és értékeit mutatja*

```
procedure TForm1.ComboKeyChange(Sender: TObject);
begin
  case ComboKey.ItemIndex of
    0: Reg.RootKey := HKEY_CLASSES_ROOT;
    1: Reg.RootKey := HKEY_CURRENT_USER;
    2: Reg.RootKey := HKEY_LOCAL_MACHINE;
    3: Reg.RootKey := HKEY_USERS;
    4: Reg.RootKey := HKEY_CURRENT_CONFIG;
    5: Reg.RootKey := HKEY_DYN_DATA;
  end;
  Reg.OpenKey ('\', False);
  UpdateAll;
  ComboLast.Items.Clear;
end;
```

Az új gyökérkulcs beállítása után a program megnyitja annak gyökérelemét, frissíti a felhasználói felületet, és kiüríti a második kombinált listát. Ez utóbbi az utoljára kiválasztott elemeket tartalmazza, segítségével körbejárhatjuk a fát, fel-le lépkedhetünk anélkül, hogy minden alkalommal a gyökértől kellene indulnunk. Íme a kód:

```
procedure TForm1.ComboLastChange(Sender: TObject);
begin
  Reg.OpenKey (ComboLast.Text, False);
  UpdateAll;
end;
```

Ez a kód egyszerű, viszont az ezen kombinált listában szereplő elemek listájának frissítéséhez szükséges programrész már meglehetősen bonyolult. Amellett, hogy ellenőriznünk kell, hogy egy adott elérési út már létezik-e, minden olyan elérési út elejére „\"-t kell tennünk, melyben az kezdetben nem szerepelt. A problémát itt az okozza, hogy a fenti eljárás hatása kifogástalan, a Reg objektum CurrentPath tulajdonságát azonban nem állítja be megfelelően. Ahhoz, hogy onnan további kiválasztásokat végezhessünk (egyszer már hozzáadtuk a listához), először ezt korrigálnunk kell. Mindezeket figyelembe véve elkészült a ListSubClick metódus végső változata:

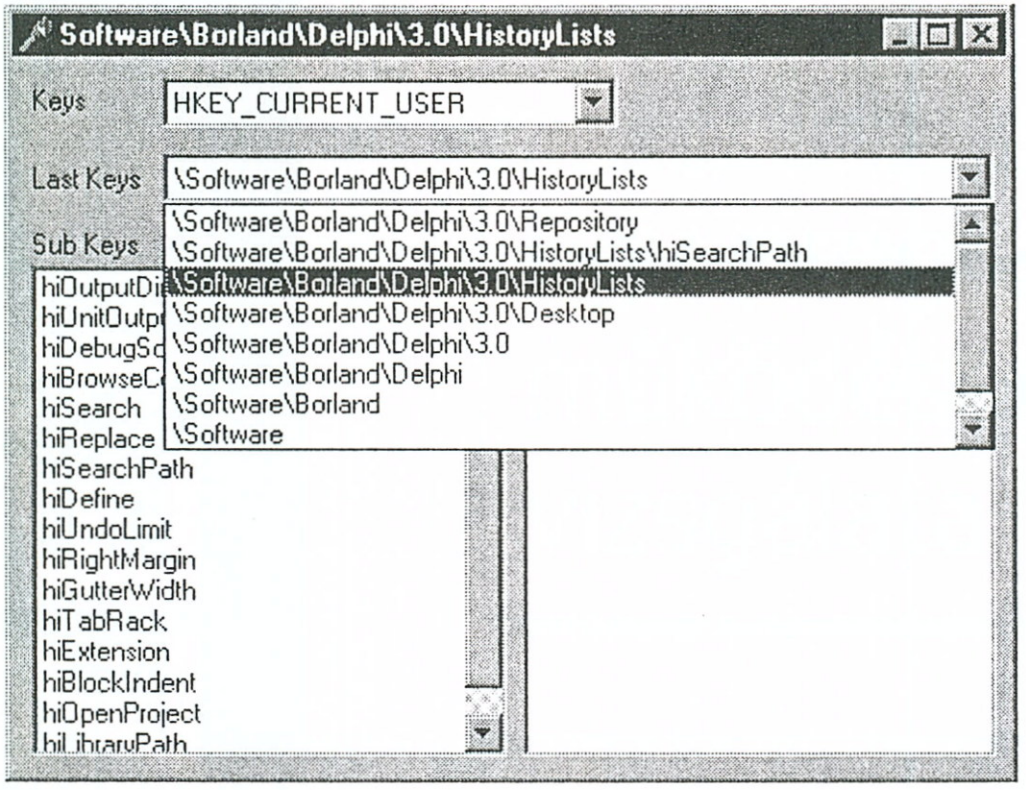
```
procedure TForm1.ListSubClick(Sender: TObject);
var
  NewKey, Path: string;
  nItem: Integer;
begin
  // get the selection
  NewKey := ListSub.Items [ListSub.ItemIndex];
  Reg.OpenKey (NewKey, False);
  // save the current path (eventually adding a \)
  // only if the it is not already listed
  Path := Reg.CurrentPath;
  if Path < '\' then
    Path := '\' + Path;
  nItem := ComboLast.Items.IndexOf (Path);
  if nItem < 0 then
  begin
    ComboLast.Items.Insert (0, Path);
    ComboLast.ItemIndex := 0;
  end
  else
    ComboLast.ItemIndex := nItem;
  UpdateAll;
end;
```

A 25.18. ábrán a kombinált listához hozzáadott elemekből láthatunk néhányat.

Hogyan tovább?

Ebben a fejezetben megismerhettük az Application globális objektum néhány jellemzőjét, szó volt párhuzamos programozásról, időzítőkről és háttérszámításról. Emellett megvizsgáltuk a rendszer által a programoknak átadott paraméterek fontosságát, és nagyon rövid programokat készítettünk kevés komponenssel, sőt, komponens nélkül.

A példák megmutatták, hogy a rendszer nagyon alacsony szintű jellemzőinek használatával a Delphi lehetőséget ad a Windows programozásának bonyolultabb részeibe való elmélyedésre.



25.18. ábra *A RegView példa a legutóbb használt Registry kulcsainak listájával*

A Delphi nagyszerű tulajdonsága, hogy segítségével a Windows összes lehetőségét kihasználhatjuk. El kell ismernünk, hogy néha az egyes alacsonyszintű feladatok megírására a tiszta C kód egyszerűbb, de ez csak azért van, mert a rejtett alkalmazás-ablak jelenléte némi kellemetlenséget okoz. Eddig azonban még semmi olyannal nem találkoztam, amit C programozással meg tudtam volna oldani, de Delphiben nem. A Delphi rugalmassága szokatlan egy vizuális programozói környezettől.

Tanultunk az Application objektumról, de az alkalmazás teljes szerkezetét még mindig nem tártuk fel. A következő két fejezet, melyek olyan témákról szólnak, mint a memória és az erőforrások, teljessé teszi a képet.

26. FEJEZET

Egy program működésének vizsgálata

E fejezet tartalma:

- A hibakereső (debugger) használata
- Hibakeresés, másképpen
- Egy futtatható állomány vizsgálata az Object Browserrel
- Az üzenetvezérlés elemzése
- Az alkalmazás a memóriában
- A Windows rendszermemóriája

Egy program működésének vizsgálata

Egy Delphi program sikeres megírása és lefordítása után joggal gondolhatjuk azt, hogy készen vagyunk feladatunk megoldásával, ám a legtöbb esetben a programok még tartalmazhatnak olyan hibákat, amelyekre csak azok futása közben derül fény. Ekkor a problémák kijavítása előtt az első lépés annak a megtalálása, hogy a program mely része vagy részei nem működnek a kívánt módon, vagy adnak nem megfelelő eredményt. Egy nagyobb programnál ennek megtalálása hosszadalmas, idegőrlő feladat. Szerencsére nagyon sok olyan eszköz és lehetőség áll rendelkezésünkre, amelyekkel egy Windows alkalmazás viselkedését tanulmányozhatjuk.

A Delphi fejlesztőrendszer tartalmaz egy beépített hibakeresőt és egy Object Browser nevű alkalmazást, amelyekkel a fordítási folyamat eredményét kétféleképpen értékelhetjük ki. Annak érdekében, hogy megérthessük azt, hogy mi is történik egy programban, nem elég csak annyit tudnunk, hogy hogyan működnek ezek az eszközök, azzal is tisztában kell lennünk, hogy a Delphi és maga a Windows operációs rendszer milyen úton-módon használja a memóriát. Ebben a fejezetben pontosan ezekről olvashatunk, s megértésüket egyszerű példák segítik. Először a hibakeresőt, majd az Object Browsert fogjuk behatóbban tanulmányozni. Ezek a részek nem mutatják be teljes részletességgel a programok összes funkcióját, csupán technikai oldalról közelítik meg a témát. A fejezet második részében a Windows 95 operációs rendszer memóriahasználatáról, illetve a Delphi programok memóriába való leképzéséről olvashatunk bővebben.

A hibakereső (Debugger) használata

Ahogyan azt már tudjuk, egy Delphi program fejlesztői környezetben való futtatáskor valójában a beépített hibakereső futtatja a kódot (ennek beállításait az Environment Options párbeszédablak **Preferences** lapján, az **Integrated Debugger** opción keresztül tudjuk megváltoztatni).

Miközben egy program a hibakeresőn belül fut, bármikor felfüggeszthetjük annak futását, ha a SpeedBar **Pause** gombjára kattintunk. Ha a programot ily módon felfüggesztettük, akkor annak futtatását lépésről lépésre bármikor folytathatjuk, ha a **Step Over** feliratú gombra kattintunk. Ugyanezt a gombot kell használnunk, ha a programot a legelejétől, lépésekben szeretnénk végrehajtatni. Azt azonban ne felejtsük el, hogy mint minden Windows program, a Delphi alá írt programok is üzenetvezéreltek, így teljes egészében nem lehet őket lépésről-lépésre végrehajtatni, mint ahogyan azt a DOS-os programokkal tehetjük. Ebből kifolyólag a legtöbbször használt megoldás egy Delphi vagy más Windows program hibakeresésében a töréspontok (breakpoint) felállítása a program azon részeiben, ahol a hibát sejtjük.

Hibakeresés DLL-ben és ActiveForm-ban

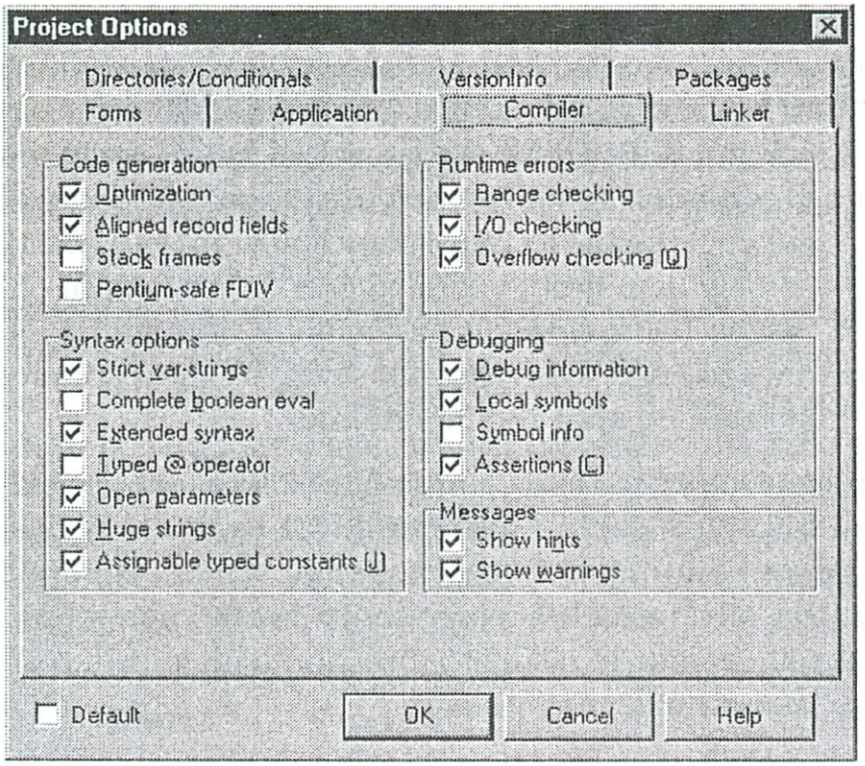
A Delphi 3-ban a beépített hibakeresőt használhatjuk DLL-ek vagy más könyvtárak, mint például ActiveX vezérlők, hibakereséséhez. Egyszerűen csak nyissuk meg a könyvtár Delphi projektjét, majd válasszuk a **Run | Parameters** parancsot, majd adjuk meg a Host Application, vagyis az azt futtató alkalmazás nevét.(ez az opció csak akkor aktív, ha az aktuális projekt egy könyvtár). Ha most megnyomjuk az F9 funkcióbillentyűt, vagy a **Run** gombot, akkor a Delphi elindítja a végrehajtható állományt, amelynek viszont be kell töltenie a könyvtárat. Ettől a ponttól kezdve a töréspontok elhelyezése és használata a szokásos módon történik.

Ezt a módszert alkalmazhatjuk akkor is, ha ActiveForm-ot elemzünk. Ekkor a gazdaalkalmazásnak a gépen található böngészőprogramot kell megadni, (például C:\Program Files\Netscape\Netscape.exe) Run paraméterként pedig egy HTML oldalt. Hogy ez működjön, ugyancsak ki kell adni a **Run | Register** ActiveX Server parancsot. Ezután böngészőnk az általunk megadott ActiveX verziót fogja használni az OCX gyorsítótárban található egyéb verzió helyett.

Hibakeresési információ

Annak érdekében, hogy egy Delphi programon a hibakeresőt (Debugger) használhassuk, a fordítás során engedélyezni kell a hibafordítási információknak a futtatható kódhoz való hozzáfűzését. Ezt a Delphi a gyári beállítások szerint megteszi, ám ha úgy gondoljuk, hogy nincsen rá szükség, akkor a Project Options párbeszédablakon keresztül bármikor ki-, illetve bekapcsolhatjuk. Ahogyan azt a 26.1. ábrán láthatjuk, a **Compiler** oldal három jelölőnégyzetet tartalmaz.

- Ha a **Debug Information**-t választjuk, akkor minden egységhez egy lista fűződik, amely tartalmazza a futtatható kódok helyét és a forráskódban nekik megfelelő sor számát.
- A **Local Symbols** választásával az osztályokban, eljárásokban, funkciókban és metódusokban található azonosítókról készül hibakeresési információ.
- A **Symbol Info** bekapcsolása esetén kereszthivatkozási tábla készül a modulban található szimbólumokról, így azokat az Object Browser segítségével már megjeleníthetjük.



26.1. ábra *Használjuk a Project Option párbeszédablak Compiler oldalát a hibakeresési információk futtatható állományhoz való fűzésének engedélyezéséhez*

Ezeket az információkat a hibakereső használja fel a program ellenőrzése során, s mindaddig nem lesznek a futtatható állományhoz csatolva, ameddig a Project Option párbeszédablak **Linker** oldalán található **TD32 Debug Info** kapcsolót be nem állítjuk. Bekapcsolása előtt ne feledkezzünk meg arról, hogy az állományhoz csatolt hibakeresési információknak két nagy hátránya van: megnöveli a futtatható állomány méretét, illetve lehetővé teszi más személynek, hogy megfelelő hibakereső birtokában kielemezhesse a programunk működését. Ezért csak akkor adjunk ilyen információkat a futtatható állományhoz, ha a programot más hibakeresővel, például a Borland cég Turbo Debugger for Windows (TDW32) termékével szeretnénk használni. Lehetőleg ne adjuk hozzá, ha csak a beépített hibakeresőt használjuk, vagy a programot már a megrendelőnek szállítjuk. Hihetetlennek tűnhet, de még most is sok olyan program van a piacon, amely tartalmaz hibakeresési információt.

A töréspontok beállítása

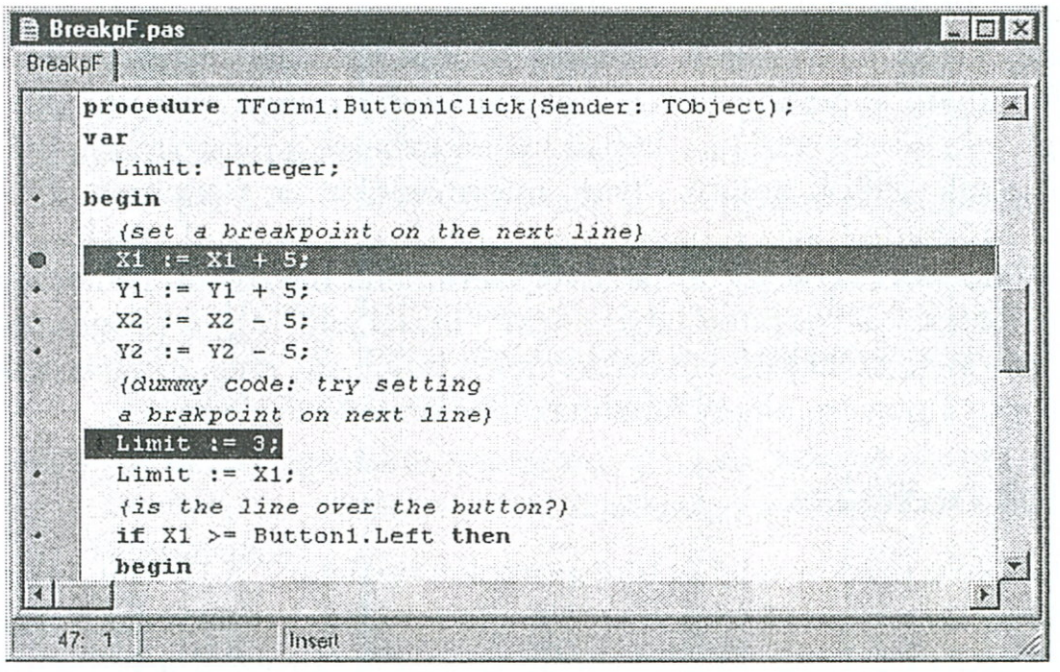
A Delphiben többféle módon is lehet töréspontokat beállítani. A Delphi 3-ban a legegyszerűbben a szerkesztőablak margójára (a forráskód és az ablak szélei között lévő terület) való kattintással tehettük meg. Miután azon a területen egyet kattintunk, az egér a sorhoz közel tűnik fel, illetve a sor megváltoztatja a színét. Az ekkor megjelenő picike pontok jelölik azokat a helyeket, ahol a töréspontokat felállíthatjuk.

Nem állíthatunk fel töréspontokat csak úgy, bárhol a forráskódban. Alapvetően a töréspont csak akkor hatásos, ha olyan sorban helyezzük el, amelyből a fordítás végén olyan futtatható kódrészlet keletkezik, amely valamikor a program élete során, végrehajtásra is kerül. Ez pontosabban azt jelenti, hogy töréspontokat nem helyezhetünk el, s nincs is értelme elhelyezni a forráskódban található megjegyzésekben, deklarációkban, fordítási kapcsolók megadásánál, illetve bármely olyan sorban, amely nem kerül végrehajtásra. Bár a Delphi engedélyezi, hogy tetszőleges helyen töréspontokat használjunk, mindig figyelmeztetni fog minket a töréspont színének megváltoztatásával, ha a töréspontot a forráskód olyan pontján helyeztük el, ahol semmi értelme.

Itt kell megjegyeznünk, hogy a Delphi fordítója a futtatható állományt optimalizálja, így azokat a sorokat a fordítás során figyelmen kívül hagyja, amelyek megjegyzést tartalmaznak, nem kerül rájuk a vezérlés, vagy semmi szerepük nincsen a program logikájában. Mivel ezekből a sorokból a fordító nem képez végrehajtható programrészletet, ezért az itt található töréspontokat a fordító érvénytelennek jelölheti. Erre lássunk most egy példát:

```
Limit := 3;
Limit := X1;
```

Az első sornak nincsen hatása a második sorra, sem az utána következő programrészletre, így a fordító kódoptimalizáló része az első sort figyelmen kívül hagyja. Ebből következik, hogy a töréspont beszúrása esetén az első sor alatt nem jelennek meg a pici pontocskák, azt jelképezve hiányukkal, hogy az első sornak semmilyen hatással nincsen a program futására (26.2. ábrát).



26.2. ábra *Ennek a példaprogramnak néhány sorát a fordító figyelmen kívül hagyja. Azokra a sorokra, amelyek nem tartalmaznak kis pöttyöket a margón, nem lehet töréspontot állítani*

Mindebből az következik, hogy ha olyan érvénytelen helyen hozunk létre töréspontot és a programot lépésenként hajtatjuk végre, akkor a hibakereső azt a töréspontot figyelmen kívül hagyja, hiszen a sor, amelyre a töréspont hivatkozik, a fordítás során nem került bele a futtatható állományba. Csak néhány speciális esetben, mint például ennél, áll meg a program futása.

A töréspontok használatával főleg azt ellenőrizhetjük, hogy bizonyos eseménykezelők a megfelelő módon futottak le vagy sem. Amennyiben bármilyen kétségünk merülne fel azzal kapcsolatban, hogy a program rendesen fut-e, vagy egy-egy megszakítás-kezelő meghívása a valóságban tényleg megtörtént-e, minden egyes eljárás elé töréspontok beszúrásával könnyedén ellenőrizhetjük. Erre mutat egy remek példát a BreakP példaprogram, amely saját formja köré keretet rajzol. A form közepén található gombbal a vonalakat a form közepe felé mozgathatjuk, de sohasem írhatjuk felül a gombot velük. A FormPaint eljárás négyszögeket rajzol egyenesek sorozatával:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  {set a breakpoint on the next line}
  Canvas.MoveTo (X1, Y1);
  Canvas.LineTo (X2, Y1);
  Canvas.LineTo (X2, Y2);
  Canvas.LineTo (X1, Y2);
  Canvas.LineTo (X1, Y1);
end;
```

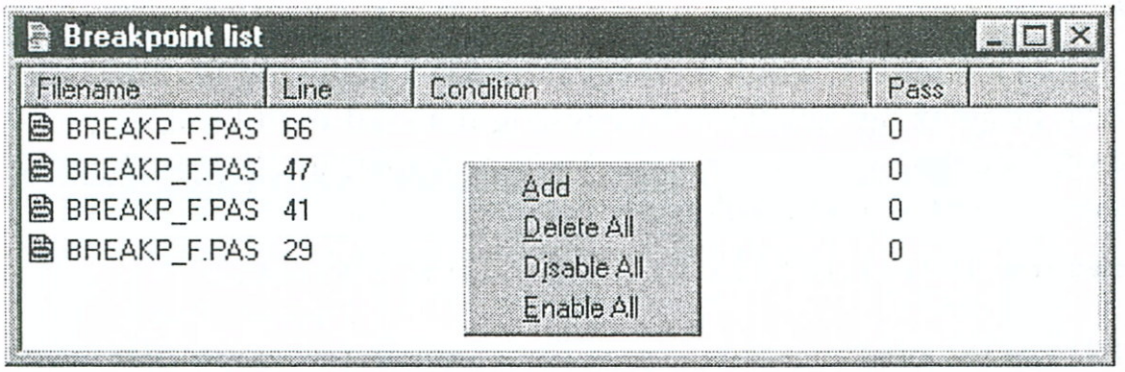
Amikor a gombra kattintunk, akkor az megváltoztatja a négyszög koordinátáit, majd ellenőrzi, hogy az így keletkezett négyszög nem lett-e túl kicsi, vagy átfedi a gombot. Végül a program érvényteleníti a formot, így hagyja, hogy a Windows a `wm_Paint` üzenete küldje el:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Limit: Integer;
begin
  {set a breakpoint on the next line}
  X1 := X1 + 5;
  Y1 := Y1 + 5;
  X2 := X2 - 5;
  Y2 := Y2 - 5;

  {dummy code: try setting a breakpoint on next line}
  Limit := 3;
  Limit := X1;

  {is the line over the button?}
  if X1 >= Button1.Left then
  begin
```

```
        Button1.Enabled := False;
        X1 := Button1.Left;
      end;
      if Y1 >= Button1.Top then
      begin
        Button1.Enabled := False;
        Y1 := Button1.Top;
      end;
      Invalidate;
    end;
```

Amikor a felhasználó átméretezi a formot, akkor a program a négyzet koordinátáit alaphelyzetbe állítja:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  {set a breakpoint on the next line}
  Button1.Enabled := True;
  X1 := 10;
  Y1 := 10;
  X2 := ClientWidth - 10;
  Y2 := ClientHeight - 10;
  Invalidate;
end;
```



26.3. ábra *A Breakpoint List ablak, saját helyi menüjével*

Ha a megjegyzések utáni sorokban töréspontot helyeztünk el, akkor már a **View** menüben található **Breakpoints** paranccsal megjeleníthetjük meg a Breakpoint List ablakot, amely az általunk elhelyezett töréspontokat sorolja fel (26.3. ábra).

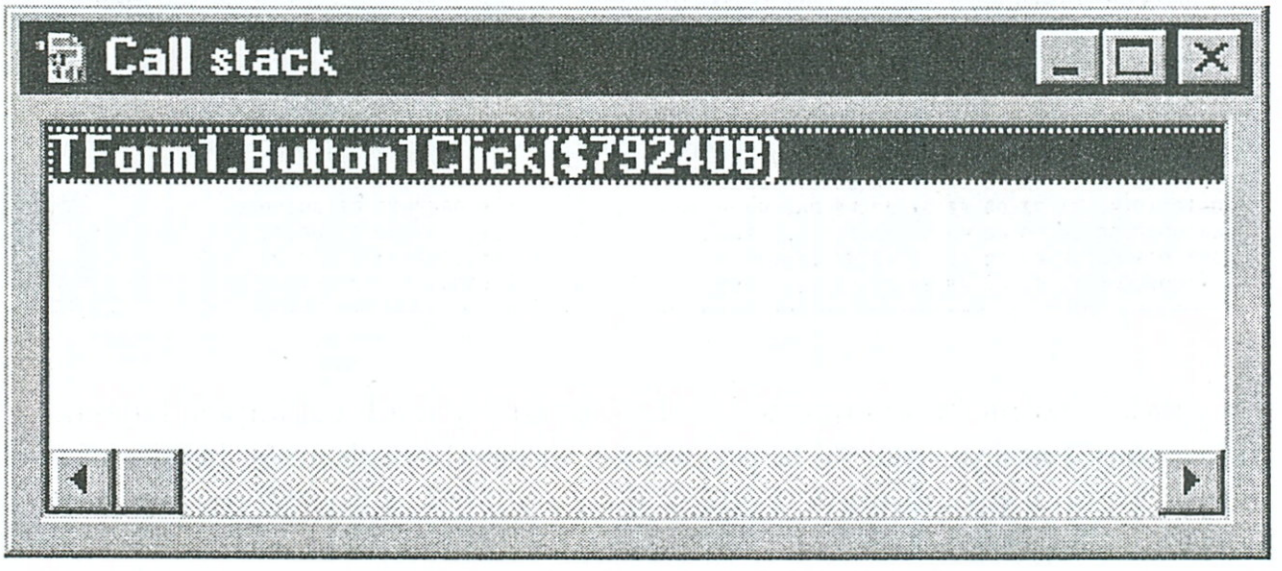
A töréspontokhoz feltételeket is rendelhetünk, így elérhetjük azt, hogy a program futása csak azoknál a töréspontoknál szakad meg, amelyekre a feltétel teljesül. Ezekről kicsit később, de még ebben a fejezetben olvashatunk.

Ha a program futása egy töréspontnál megszakad, akkor azt bármikor folytathatjuk azt a **Run** paranccsal. Mint ahogyan arról korábban szót ejtettünk, a programokat lépésenként is lehet futtatni. A SpeedBar-on található **Step Over** gomb segítségével a program szakaszonként hajtatható végre, míg a **Trace Into** gombra kattintva megtudhatjuk, hogy éppen mely eljárások lettek meghívva és végrehajtva.

Azt a sort, amelyet a Delphi éppen végrehajt, más színnel és egy kis nyilacskával jelöli, így egyből láthatjuk, hogy hol tart programunk végrehajtása.

Figyeljük meg, hogy ha a végrehajtás során egy másik egységbe kerülünk, akkor annak a forráskódja egy új szerkesztőablakban jelenik meg, amennyiben lehetséges. A program futásának követése közben a meghívott szubrutinok sorozatát és egymásba ágyazódását a Delphi **View** menüjében található **Call Stack** paranccsal nézhetjük meg. Ha egy gomb OnClick eseményéhez töréspontot rendelünk, akkor a program megállítása után a **Call Stack** gombra kattintva a 26.4. ábrához hasonló eredményt láthatunk a képernyőn.

Itt láthatjuk az eljárások neveit és memóriacímeit, amelyek hasznosak lehetnek az egymásba ágyazott szubrutinokban való hibakereséshez.

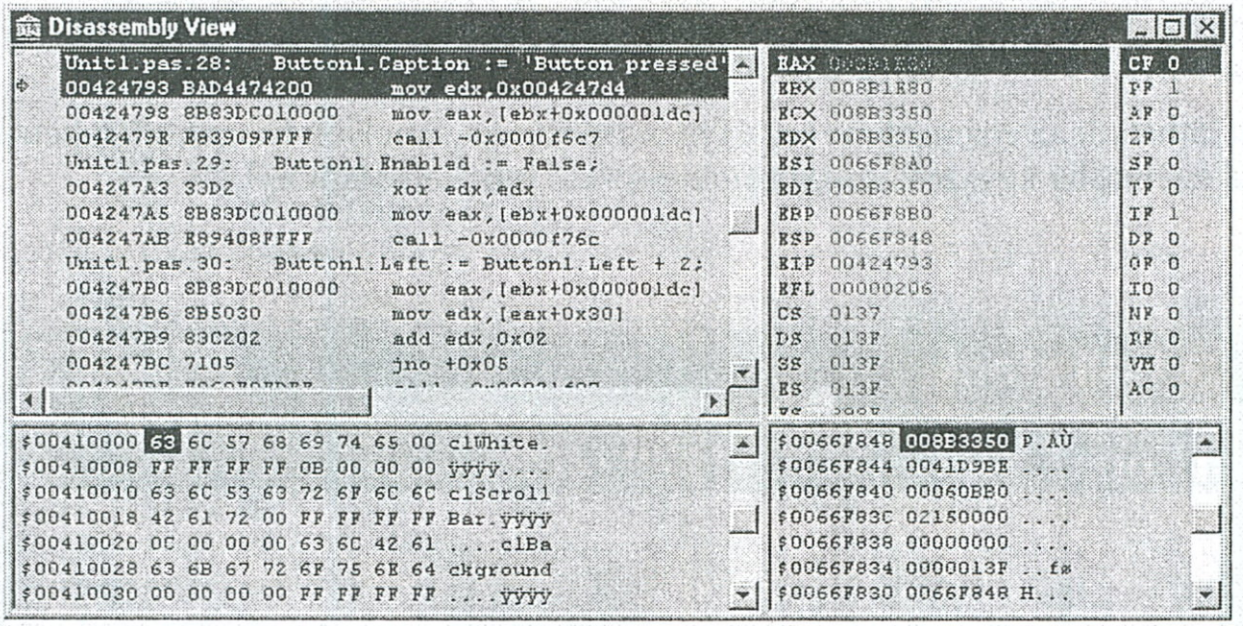


26.4. ábra *A Call Stack ablak, egy gomb megnyomása után*

A CPU ablak, avagy a Disassembly View

Egy másik Delphi 3-ban használható hibakeresési ablak a CPU View. Ennek engedélyezéséhez egy bejegyzést kell elhelyeznünk a Regisztrációs adatbázisban. Menjünk a HKEY_CURRENT_USER bejegyzéshez, majd keressük meg a Borland\Software\Delphi\3.0\Debugging albejegyzést.

Ez alá készítsünk egy `Enable CPU` bejegyzést, s értékként állítsunk be egyet. Ha legközelebb a Delphi futtatjuk, akkor egy új menü jelenik meg, a **View | CPU Window**. Ennek használatával a hibakeresés közben rengeteg rendszerinformációt, köztük a CPU regisztereit, jelzőit, és még sok minden mást követhetünk nyomon. Íme egy példa ennek az ablaknak a kinézetére:



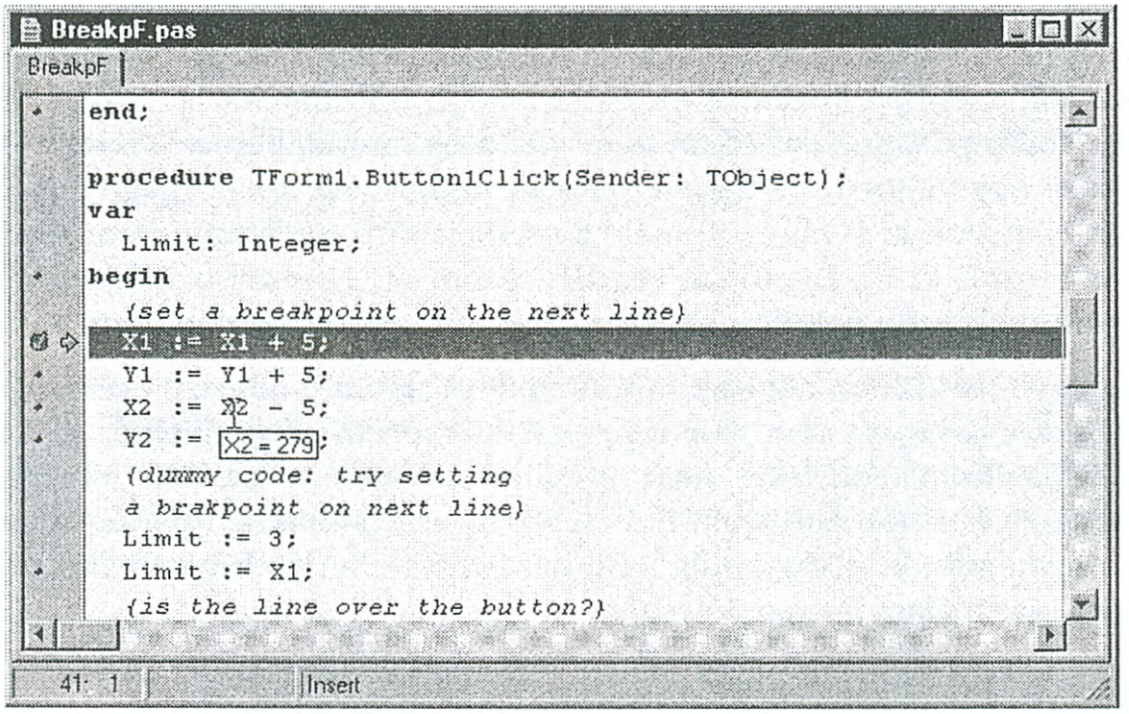
Ha rendelkezünk némi tapasztalattal az assembly nyelvvel kapcsolatosan, akkor ebből az információból pontos képet nyerhetünk arról, hogy mit is csinál éppen a programunk. Például megnézhetjük a fordító optimalizáló hatását a programunkon.

Bár a Disassembly View ablak első ránézésre elég szokatlannak tűnhet, rengeteg lehetőséget tartalmaz. Előugró menüivel a CPU regisztereit akár közvetlenül is megváltoztathatjuk. Azonban azt jegyezzük meg, hogy ez a funkció hivatalosan nem dokumentált, így ha valami nem működik úgy ahogyan azt szeretnénk, akkor annak megoldásában nem kaphatunk támogatást.

Az értékek vizsgálata

A program futtatásának megállításakor az azonosítóhoz (változók, objektumok, komponensek, tulajdonságok és így tovább) tartozó értéket ellenőrizhetünk. A Delphi korábbi verzióiban erre alapvetően két módszer volt ismert: az Evaluate/Modify párbeszédablak használatával, vagy a Watch List ablakban egy bejegyzés elhelyezésével. A későbbiekben még foglalkozunk ezekkel.

A Delphi 3 ezzel szemben tartalmaz egy úgynevezett „fly-by" módszert, amelynek lényege az, hogy a program futásának megállítása után, ha egy azonosítóra kattintunk, akkor annak értéke egy kis ablakban megjelenik nem messze az egérkurzortól. Erre láthatunk egy példát a 26.5. ábrán.



26.5. ábra *A legjobb Delphi 3 hibakereső funkció: a „fly-by" üzenetek*

Egyszerű változók esetében, mint például X1 vagy Y1, a „fly-by" ablakocska csak az értéket mutatja meg. Joggal merülhet fel a kérdés, hogy mi van akkor, ha például azt szeretnénk megtudni, hogy mi az értéke egy Form1 vagy Button1 objektumnak? Ezekben az esetekben azt fogjuk tapasztalni, hogy a Form1-hez hasonló objektumok esetében az ahhoz rendelt mezőket és értékeit kapjuk vissza, míg a Button1 ablakocskája csak néhány, igen ritkán használt információt fog visszaadni. Az alapvető probléma az, hogy a Delphi alkotóelemeinek, komponenseinek olyan sok tulajdonságuk van, hogy azt nem lehet, de nem is célszerű mind felsorolni.

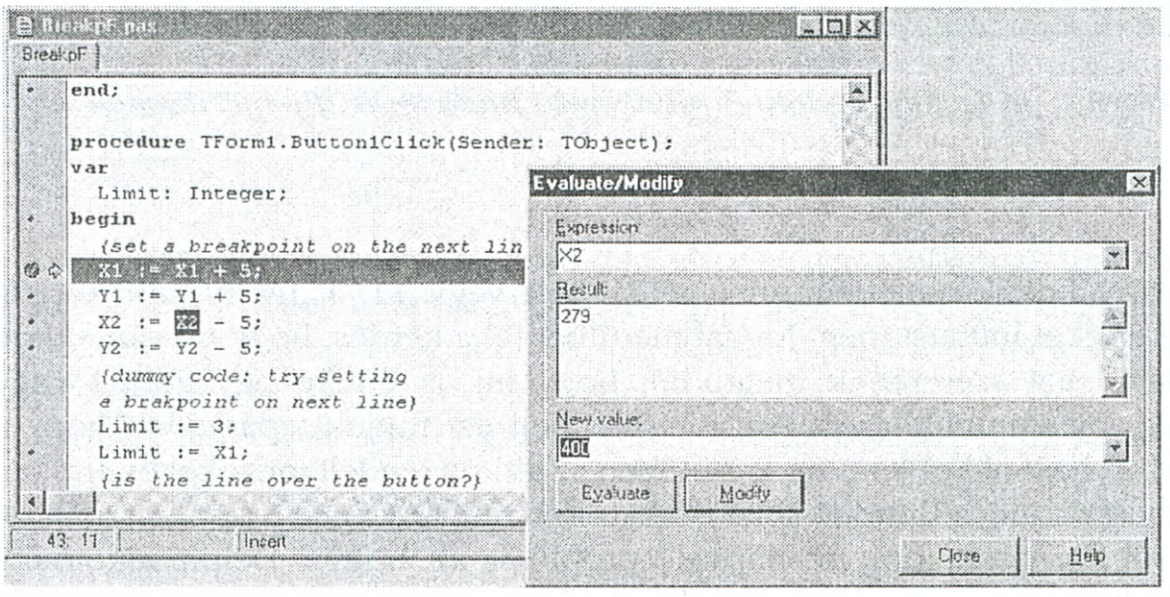
Tipp

A mellékelt CD-n található az általam által írt Object Debugger alkalmazás, amely egy komponens futási ideje alatt megmutatja a komponens összes létező tulajdonságának értékeit.

Más esetekben az így kapott érték meglehetősen furcsa is lehet. Például sorszámozott típusoknál kaphatunk egy „True=True" kifejezést is. Viccen kívül, nagyon sok érdekes kifejezést ellenőrizhetünk ennek segítségével. Ha például egy színkonstansra bökünk, legyen ez mondjuk a clYellow, akkor annak számszerűsített értékét kapjuk vissza.

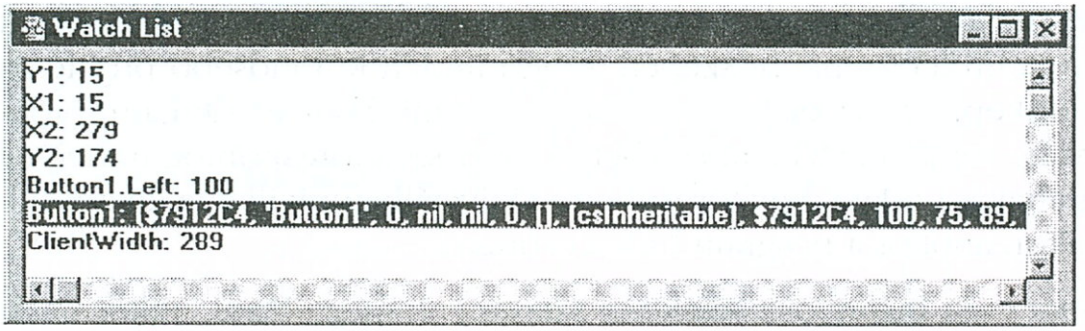
Az előbbi technikával szembeállítva a fentebb említett másik két módszert, azok sokkal nehezebben kezelhetőek és kevesebb információt is nyújtanak. Az Evaluate/Modify párbeszédablakon keresztül még mindig képesek vagyunk megváltoztatni egy változó értékét vagy tulajdonságát. Az ablak megnyitásához először válasszuk ki a változót, majd a szerkesztő SpeedMenu menüjéből (CTRL-F7) válasszuk ki az **Evaluate/Modify form**-ot. Hosszabb kifejezések esetén érdemesebb azt egy külön ablakba kimásolni, és ott szerkeszteni.

Például, ha egy töréspontot állítunk rá a BreakP program OnResize eseménykezelőjére, akkor a programon való lépkedés közben megfigyelhetjük az X1 értének módosulásait, sőt, akár meg is változtathatjuk tetszésünk szerint, ahogyan azt a 26.6. ábrán láthatjuk. Ebben a párbeszédablakban bármekkora kifejezést megadhatunk, addig, amíg nem használunk benne függvényhívást. A konstansok használata is engedélyezett.



26.6. ábra *Az Evaluate/Modify párbeszédablakon keresztül a változók értékeit vizsgálhatjuk vagy változtathatjuk meg*

Amikor változók csoportjának értékét akarjuk ellenőrizni újra és újra, például cikluson belül, akkor a fly-by módszer egy kicsit monoton és fárasztó lehet. Ilyen esetekben inkább állítsunk be nézőkében azokat a változókat, tulajdonságokat vagy komponenseket, amelyeket figyelemmel szeretnénk kísérni. Ezzel például figyelemmel kísérhetjük a BreakP program Button1Click eljárásának – amely akkor hajtódik végre, ha a Button1 gombra kattintunk – minden egyes értékét. A 26.7. ábrán láthatjuk, hogy a nézőkében az eljárás legfontosabb paramétereinek követése lett beállítva.



26.7. ábra *A Watch List ablak használata*

Új nézőke beállításához használjuk a szerkesztő SpeedMenu-jében található **Add Watch at Cursor** parancsot, vagy a CTRL-F5 billentyűkombinációt. A meghatározásakor meg kell adnunk a kimenet megfelelő formátumát, esetleg gépelni egy keveset, ha összetettebb kifejezésről van szó. Egyszerű értékek esetén az azonosító kiválasztása után csak a parancsot adjuk ki.

Egy változó értékét csak akkor láthatjuk, ha a program futtatása megállt vagy szünetel, de sohasem akkor, amikor fut. Ezen kívül, csak azokat a változókat lehet megtekinteni, amelyek elérhetőek, azaz adott futtatási egységben léteznek. Ugyancsak adhatunk nézőkét hozzájuk, bár az ezek által nyújtott információkat gyakran nem is olyan könnyű értelmezni. Sokkal egyszerűbb és tanácsosabb, ha a komponensnek csak egy tulajdonságát követjük figyelemmel.

Bővebben a töréspontokról

Amikor a program a futás közben egy törésponthoz ér, akkor a Delphi felfüggeszti annak futását, majd megmutatja egy szerkesztőablakban, hogy melyik sor hajtódott végre utoljára. Amikor ez az ablak a képernyőre kerül, előfordulhat, hogy a program ablakának egy részét, vagy teljes egészét eltakarja. Ekkor elképzelhető, hogy ha a futtatott program ír a képernyőre, akkor a megjelenő ablak eltakarhatja azt. Ezért, lehetőleg a szerkesztőablakot és a program futási képernyőjét úgy helyezzük el a képernyőn, hogy ne fedjék át egymást, így elkerülhetjük az előbb említett problémát.

Ez különösen akkor hasznos, ha OnPaint eseménykezelőkkel dolgozunk. Ezt akár ki is próbálhatjuk a BreakP program segítségével. Nyissuk meg még egyszer, állítsunk egy töréspontot a FormPaint eljárásra, majd futtassuk a programot. Ha a form és a szerkesztőablak átfedik egymást, akkor töréspontok végtelen sorozatába kerülünk, hiszen ahányszor a Delphi frissíti a formot, a töréspont megállítja a program futását, megjeleníti a szerkesztőablakot, amely a form frissítését váltja ki, amely ugyanannál a töréspontnál áll meg, és így tovább.

A legegyszerűbb megoldás erre az, hogy a két ablakot úgy helyezzük el a képernyőn, hogy ne fedjék egymást. Ebben az esetben láthatjuk, ahogyan a FormPaint program lépésenként szépen lefut, minden különösebb probléma nélkül. Hasonlóképpen, ha csak azt szeretnénk tudni, hogy az OnPaint program lefutott vagy sem, akkor kikapcsolhatjuk az összes töréspontot, mihelyt a program végrehajtása odaér. Egy töréspontot aktivizálhatunk illetve kikapcsolhatunk bármikor, amikor a program futása szünetel.

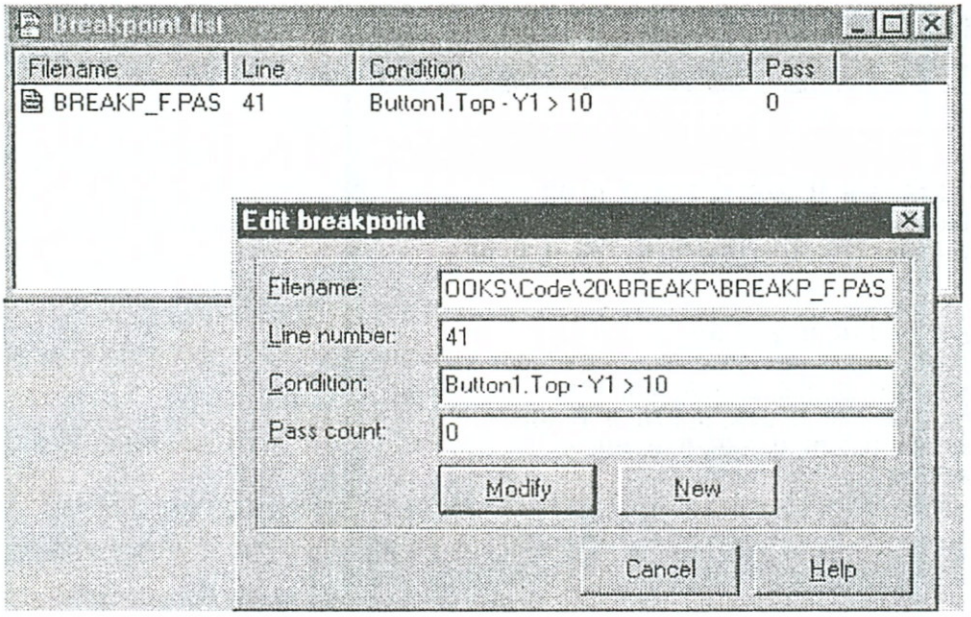
Ha a programot csak bizonyos esetekben szeretnénk megállítani, akkor használhatjuk a feltételes töréspontokat. Tegyük fel a példa kedvéért, hogy mi csak akkor szeretnénk a BreakP program Button1Click eljárását megállítani, ha a vonalak a gombhoz közel kerülnek. Állítsuk be a töréspontot a szokásos módon, nyissuk meg a Breakpoint listát, kattintsunk duplán a töréspontón, majd a megjelenő **Edit Breakpoint** párbeszédablakba gépeljük be a következő feltételt:

```
Button1.Top-Y1 < 10
```

A feltétel megjelenik a **Breakpoint List** ablakban, ahogyan azt a 26.8. ábrán láthatjuk. Ha most futtatjuk a programot, akkor párszor nyugodtan kattinthatunk a gombra, semmi nem fog történni, amíg a megadott feltétel nem teljesül (amikor a `Button1.Top-Y1` kisebb, mint 10), de akkor a hibakeresőben futó program végrehajtása megáll.

Megjegyzés

A Windowsban vannak olyan esetek, amikor az üzenetvezérelt hibakeresők, mint például a Delphi beépített hibakeresője, nem képes a program futását egy adott töréspontnál leállítani. Ebben az esetben a Windows úgynevezett „kemény" állapotban van. Ez általában menü rajzolása közben és néhány rendszeradminisztrációs tevékenység közben fordul elő. Ha a Delphi ilyen helyen talál töréspontot, akkor erre a tényre mindig figyelmeztet bennünket.



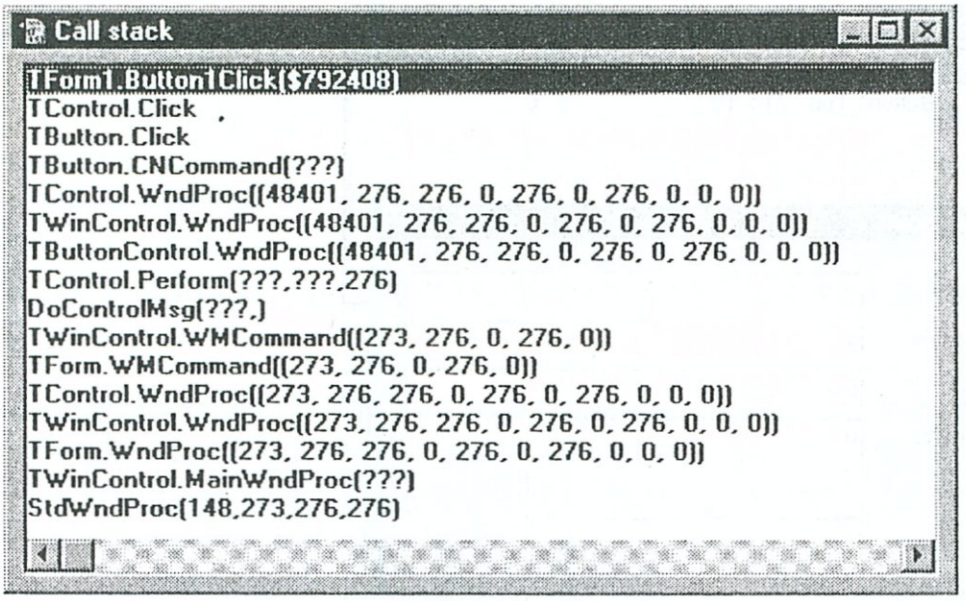
26.8. ábra *A töréspontokat tartalmazó lista, feltételes töréspontokkal*

Nyomkövetés a VCL forráskódban

Amennyiben rendelkezünk VCL forráskóddal, akkor tudnunk kell, hogy annak egyik fő felhasználási területe az alkalmazásokban való hibakeresés. Amennyiben többet szeretnénk megtudni a VCL komponens tulajdonságok eléréseinek hatásairól, mint amennyit a dokumentáció elmond, akkor fűzzük a könyvtár forráskódját a programunkhoz, majd használjuk a hibakeresőt, hogy megérthessük a működését. Természetesen ez rengeteg időt és elszántságot igényel, bár egyes esetekben, amikor minden más kudarcot vall, ez lehet az egyetlen megoldás.

A forráskód beillesztéséhez nem kell mást tennünk, mint a VCL forrását tartalmazó könyvtár nevét kell a Project Options párbeszédablak **Directories/Conditional** oldalának **Search Path** kombinált listájához hozzáadni. Ezután fordítsuk le a programot minden elemével együtt még egyszer. Ha minden jól ment, akkor használhatjuk a hibakeresőt a VCL rejtelmeinek felkutatására. Amikor a program futása közben eljárás vagy tulajdonság végrehajtására kerül sor, akkor láthatjuk, soronként, hogy mely VCL utasítás lesz aktuálisan végrehajtva. Természetesen a szokásos hibakeresési funkciók ugyanúgy működnek VCL-lel, mint a nélkül.

A 26.9. ábra a Call Stack ablakot mutatja egy gomb OnClick eseménykezelőjének érvénytelen hívását követő nyomkövetés után. Észre kell vennünk, hogy milyen sok dolog történik addig, amíg a vezérlés aktuálisan a Button1Click eljárásra kerül (a legfelső elem a Call Stack ablakban a legutolsó hívott eljárás, a kiemelt eljárás pedig majdnem a legfelül van).



26.9. ábra *A Call Stack ablak egy VCL forráskódot tartalmazó program hibakeresése közben*

Ha összehasonlítjuk a 26.9. ábrát a 26.4. ábrával, akkor észrevehetjük, hogy mennyivel több részletet és információt biztosít az egyik, mint a másik. A sok információnak van előnye és hátránya is. A sok részletből nagyon pontosan kiderül, hogy a programunk éppen mit csinál, de néha elég nehéz a töménytelen adatból értékes adatokat kihozni. Ez utóbbi esetben nem árt, ha ismerjük a VCL üzenetküldési rendszerét (VCL message dispatching mechanism).

Hibakeresés, másképpen

A töréspontok másik általános felhasználása az, hogy megtudjuk, hogy a program eljut-e egyáltalán egy bizonyos pontig. Erre azonban léteznek más módszerek is. A programunkba a ShowMessage segítségével beszúrhatunk egyszerű megjegyzéseket, amelyek a program futása során megjelennek, ha azokra a sorokra kerül a végrehajtás. Hasonlóak még a kimenet ablakba való irányítása, a címke szövegének megváltoztatása, fájlba való írás és így tovább.

Használatuknak két fő előnye van: megtudhatjuk, hogy az adott sor végrehajtódik-e valamikor, illetve futási információkat nyerhetünk anélkül, hogy a programot egyszer is meg kellene szakítani, illetve újraindítani. Hátránya, hogy beállítása és megszüntetése nem olyan könnyű, mint a töréspontok esetében.

Hibakeresés terminálablak segítségével

A Delphi által biztosított hibakeresési technikák az esetek túlnyomó többségében eredményesebben használhatók, mint a saját módszerek. A program

lépésenként való végrehajtásának lehetősége, a változók értékének nyomon követése akár egymásba ágyazott szubrutinokon keresztül – akár a VCL forráskódban is – egy nagyon hasznos, szinte nélkülözhetetlen programozói segédeszköz.

Mindez nem jelenti azt, hogy az otthoni technikákat el kellene felejteni, hiszen vannak esetek, amikor azok használata tanácsosabb. Vegyük például az utolsó fejezetben található BackProc példaprogramot. A programban olyan funkciók is találhatóak – időzítések és mások –, amelyeket a program megállítása megzavarhat. Ilyenkor segíthet, ha néhány hibakeresési információt egy másik ablakban jelenítünk meg. Ennek a megvalósítását láthatjuk a Back2 programban.

A Back2 program egy második, DebugFormnak nevezett formot is tartalmaz, amely befedi az egész kliens területet. Az egységben megtalálható egy olyan eljárás, amellyel a hibakeresési form listapaneljéhez szövegsort adhatunk:

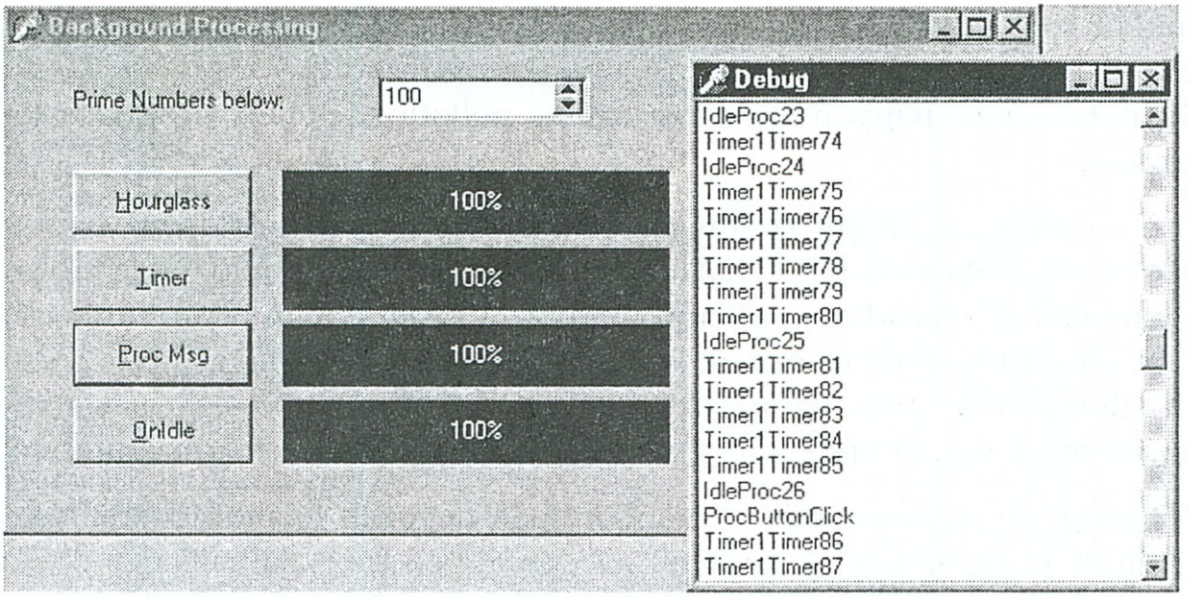
```
procedure WriteLine (Text: String);
begin
  if not DebugForm.Visible then
    DebugForm.Show;
  with DebugForm.Listbox1 do
    ItemIndex := Items.Add (Text);
end;
```

Láthatjuk, hogy ez nem egy helyi metódus, hanem egy globális eljárás, amelyet a program minden pontjáról hívhatunk. Ha ez szerepel a programunkban, akkor a form osztály minden egyes metódusában használhatjuk a WriteLine eljárást hibakereséshez úgy, hogy az nem zavarja össze a program kimeneti képernyőjét:

```
procedure TForm1.IdleButtonClick(Sender: TObject);
begin
  WriteLine ('IdleButtonClick');
  IdleNPrimes := 0;
  {same code as before...}
```

Az egyetlen hely, ahova nem lett beszúrva az előbbi programrészlet, az IsPrime funkció, ugyanis az túl gyakran kerül meghívásra. Ha most futtatjuk a programot, akkor rengeteg hibakeresési információ fog a második ablakban megjelenni. Néhány funkció a program élete során többször is végrehajtásra kerülhet, így nem árt, ha a hozzájuk tartozó hibakeresési információt sorszámmal látjuk el. Ezt végzi el a következő programrészlet:

```
procedure TForm1.IdleProc (Sender: TObject;
  var Done: Boolean);
begin
  WriteLine ('IdleProc' + IntToStr (IdleNumber));
  if IdleNumber < IdleMax then
    ...
end;
```



26.10. ábra *A Back2 példaprogram kimenete, a saját hibakereső ablakával együtt*

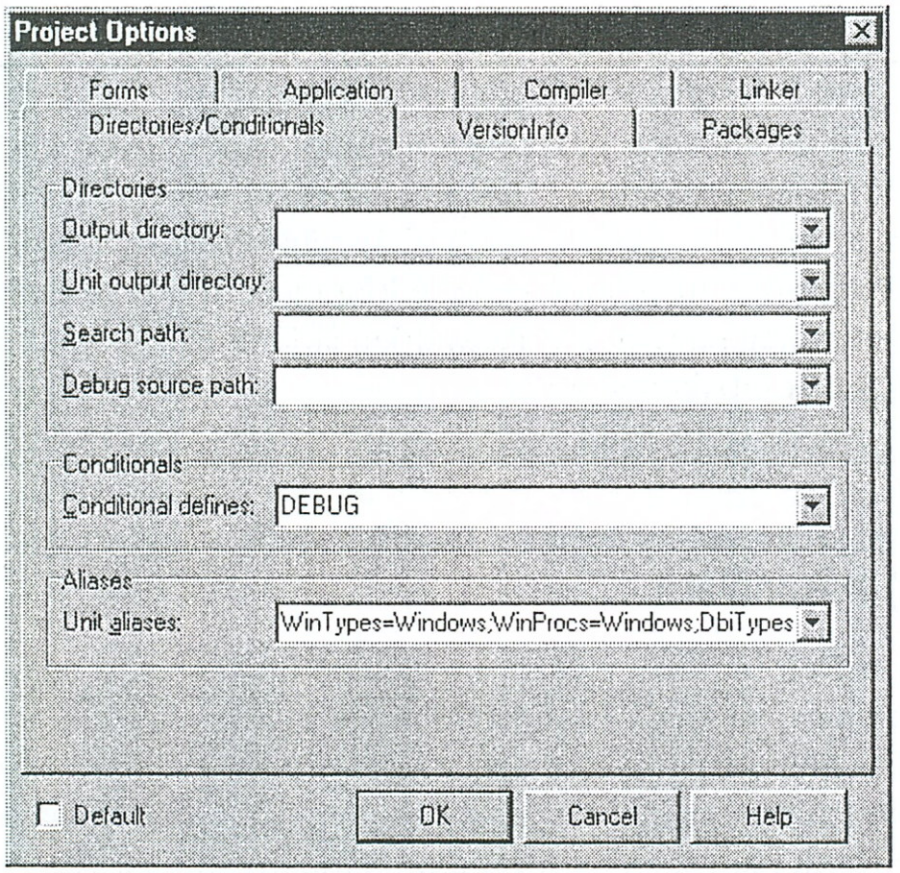
Fejlesztési és terjesztési verziók, avagy a feltételes fordítás használata

A hibakeresési információknak a programhoz való hozzáfűzése érdekes dolog és szépen is néz ki, ahogyan azt a fenti példákból is láthatjuk, de ennek ellenére a program azon verzióiban, amelyek már a felhasználókhoz kerülnek, már nem tanácsos hibakeresési információkat alkalmazni, mert azok lassítják a programot és növelik a nagyságát. Gondolhatjuk, hogy ez egy nagyon súlyos probléma és azt is, hogy a terminál ablakos megoldás amúgy sem egy professzionális megoldás. Ha már dolgoztunk valamikor C/C++-szal, akkor lehetnek ötleteink, hogy hogyan is hagyjuk el a program végső verzióiból ezeket az információkat. Az egész alapötlet a C sokat használt programozási technikájában, a feltételes fordításban gyökeredzik. Az ötlet egyszerű: bizonyos programsorokat csak akkor fordítsunk bele a programba, ha bizonyos körülmények fennállnak, más esetekben nem.

A Delphiben a következő fordítási direktívák használhatóak e célra: $IFDEF, $IFNDEF, $IFOPT, $ELSE, $ENDIF. A programunkban például kicserélhetjük a WriteLine eljárást a következőre:

```
{$IFDEF DEBUG}
WriteLine ('IdleProc' + IntToStr (IdleNumber));
{$ENDIF}
```

Ekkor ez a programrészlet csak akkor lesz belefordítva a futtatható állományba, ha a kódrészlet előtt, vagy a Project Options párbeszédablakban (26.11. ábra) megadtunk egy DEBUG szimbólumot.



26.11. ábra *Egy szimbólum létrehozása a Project Options párbeszédablakban*

Akár egy egység befordítását is szabályozhatjuk ezzel a módszerrel:

```
{$IFDEF DEBUG}
uses
  DebugF;
{$ENDIF}
```

Ha most a DEBUG szimbólumot megadjuk a Project Options párbeszédablakban, lefordítjuk a programot, majd futtatjuk azt, akkor a programunk terminál ablakot, benne hibakeresési információt is meg fog jeleníteni. Ha a DEBUG szimbólumot nem adjuk meg, akkor a megfelelő kódrészletek nem fordítódnak bele a futtatható állományba, így a terminál ablak és annak részei nem jelennek meg. A két futtatható állomány nagysága között kisebb eltérést is lehet tapasztalni, ennek az a magyarázata, hogy a második esetben a forráskód egy része a DEBUG szimbólum elhagyása miatt nem lett befordítva. Azt ugyancsak ne felejtsük el, hogy ahányszor elveszünk vagy hozzáadunk a programhoz egy olyan szimbólumot, amely befolyásolja a program fordítását, minden esetben az egész programot újra kell fordítani.

Figyelem!

Feltételes fordítási direktívákat hibakeresési célból csak nagyon különleges esetekben alkalmazzunk. Ugyanis, ha ezzel a technikával hozzuk létre a programot, majd később eltávolítjuk belőle, akkor olyan hibák jelenhetnek meg, amelyeket eddig eltakart a feltételes fordítás. Ezért lehetőleg nagyon körültekintően járjunk el a program végleges verziójának hibakeresésénél.

Állítások (assertion) használata

Az állítás elnevezés egy, a Delphi 3-ban először megjelenő hibakeresési technikát takar. Az állítás nem más, mint egy kifejezés, aminek minden körülmény között igaznak kell lennie. Állíthatjuk például azt, hogy a programunknak legalább egy felhasználó használja, hiszen a programunk nem futhat felhasználó nélkül. Ekkor, ha az állításunk hamis, akkor a programkódban – nem a végrehajtásában – valahol hibának kell lennie.

Felhasználásának érdekében a Delphi 3 bevezetett egy új eljárást, amelyet Assert-nek hívnak. Az első paramétere egy logikai kifejezés, amit tesztelni szeretnénk. Ha a logikai kifejezés igaz, akkor a program normálisan fut tovább, ha pedig nem, akkor EAssertionFailed kivétel keletkezik.

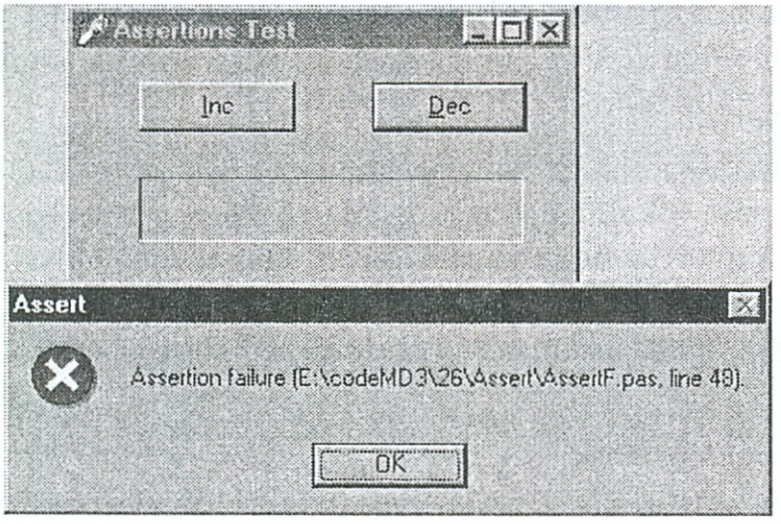
Íme egy egyszerű példa, az Assert. A formja tartalmaz egy folyamatsávot, amelynek Min tulajdonsága 1-re, Max tulajdonsága pedig 100-ra van állítva. Ezeken kívül két gomb is elhelyezésre került, amelyekkel az ugyancsak a formon található Number mező értékét lehet növelni, illetve csökkenteni, a folyamatsáv frissítésével egyetemben. A mező értékének mindig a két szélső határ között kell lennie. A programban ezt a kifejezést értékeljük minden egyes metódus végén az Assert függvény hívásával. Az Inc gomb programrészlete ezen okokból a következőképpen lett meghatározva:

```
procedure TForm1.BtnIncClick(Sender: TObject);
begin
  if Number < 100 then
    Inc (Number);
  ProgressBar1.Position := Number;
  // test the condition
  Assert ((Number > 0) and (Number <= 100));
end;
```

Ez a rész rendben működik. A Dec gombnál viszont, a példa kedvéért, elrontottam az `if` kifejezést, ezért az állítás hamissá is válhat:

```
procedure TForm1.BtnDecClick(Sender: TObject);
begin
  if Number > 0 then
    Dec (Number);
  ProgressBar1.Position := Number;
  // test the condition
  Assert ((Number > 0) and (Number <= 100));
end;
```

Ha a programot futtatjuk, s a Dec gombra kattintgatunk, akkor előbb vagy utóbb egy, a 26.12. ábrán látható kivétel keletkezik, amely arra utal, hogy az állítás hamissá vált. Az ekkor megjelenő ablakból megtudhatjuk, hogy pontosan melyik sorban található `assert` parancs kifejezése nem érte el a kívánt követelményt.



26.12. ábra *Az Assert példaprogram a Dec gombra való kattintás után, amikor az állítás hamissá válik*

Ne felejtsük el, hogy az ilyen kifejezések hibakeresési eszközök. Csak arról győződhetünk meg, hogy a forráskód rendben van. A felhasználónak sohasem szabad látnia azt, hogy ha egy ilyen állítás hamissá válik, történjen bármi is a programmal, vagy adjon neki bármilyen bemenő adatot, ugyanis az nem jelent mást, mint azt, hogy a programot elrontottuk valahol. A speciális hibalehetőségek kiszűrésére inkább kivételkezelést használjunk mintsem állításokat.

Ha az állításokat el akarjuk távolítani a futtatható állományból, akkor azt bármikor megtehetjük az $ASSERTIONS vagy a $C fordítási direktívák segítségével. Egyszerűen csak adjuk a `{$C-}` sort a nekünk tetsző egységhez, s annak futtatható kódja a legközelebbi fordítás után már nem fogja tartalmazni az állításokat. Ha erről szeretnénk megbizonyosodni, akkor a következő tesztet érdemes elvégezni: fordítsuk le az Assert példaprogramot, majd vessünk egy

pillantást a hibakeresési sorokra a szerkesztőben. Az Assert eljárást meghívó soroknak szerepelnie kell ott. Most kapcsoljuk ki az állításokat, fordítsuk újra a programot. Ha most a margóra (gutter) nézünk, akkor azt kell látnunk, hogy az előbb beállított Assert eljárásra hivatkozó bejegyzések mind érvénytelenek, ami azt jelenti, hogy nem képződött belőlük futtatható programrészlet.

Egy futtatható állomány vizsgálata az Object Browser segítségével

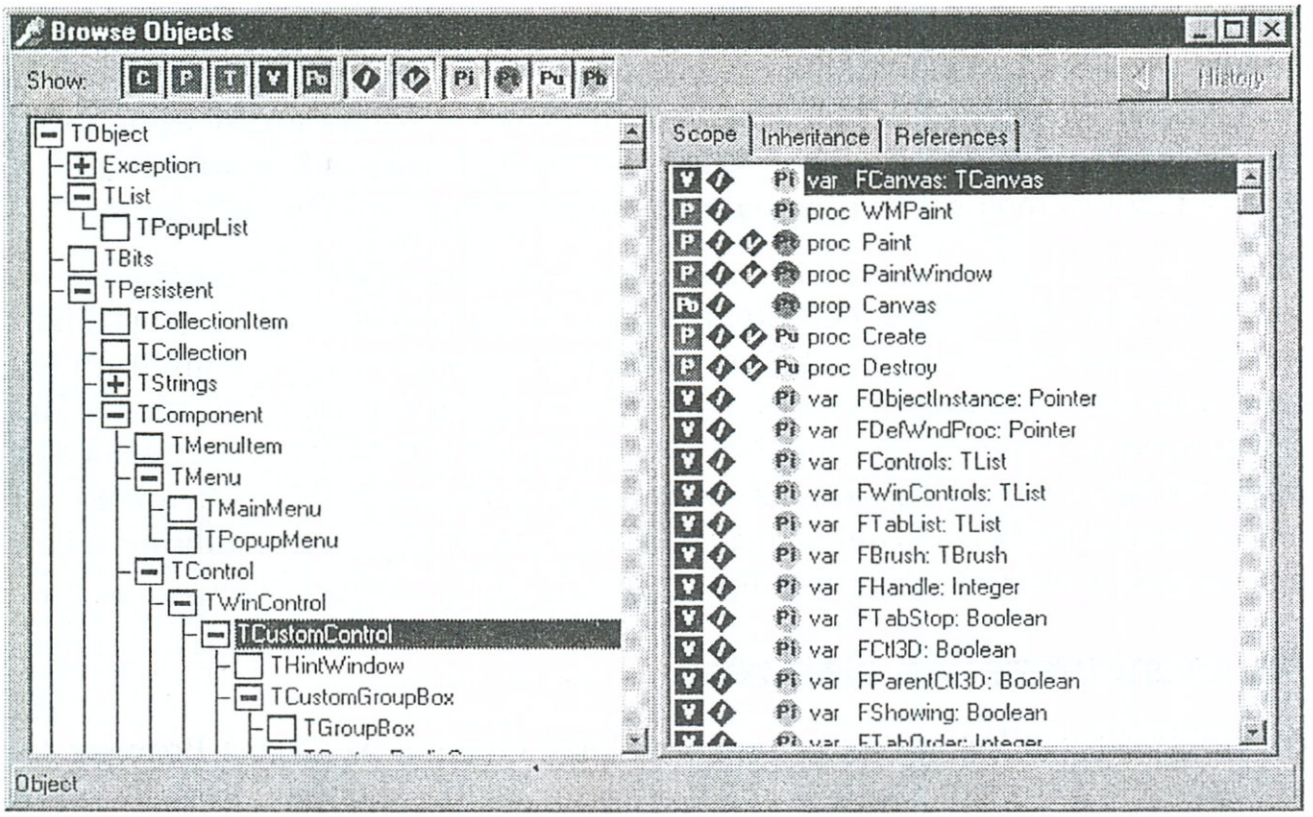
Ha egy már egyszer lefordított program működését szeretnénk vizsgálni, akkor a legmegfelelőbb eszköz számunkra a Delphi Object Browser alkalmazása lesz. A Browser nem foglalkozik az objektumok értékével vagy változóival, hanem az adattípusokat és osztályokat jeleníti meg. Ha a programunk fordítása hiba nélkül megtörtént, akkor azt már megnyithatjuk a Browser segítségével, amely rögtön megmutatja az alkalmazásunkban használt osztályok egymáshoz való viszonyát (26.13. ábra). Itt mind a VCL, mind a saját programunk által meghatározott osztályokat megtalálhatjuk.

A Delphi Object Browser alkalmazása sok tekintetben hasonló a legtöbb C++ fordítókhoz mellékelt hasonló célú alkalmazásokhoz. Alapvetően a Browser segítségével bármelyik osztály, legyen az helyi vagy örökölt, összes metódusát, tulajdonságát és adatmezőit megtekinthetjük. Ha csak bizonyos elemekre vagyunk kíváncsiak, akkor a Browse Object ablak tetején található **Filter** gombbal beállíthatjuk, hogy mely tulajdonságnak megfelelő elemek legyenek csak listázva (26.14. ábra).

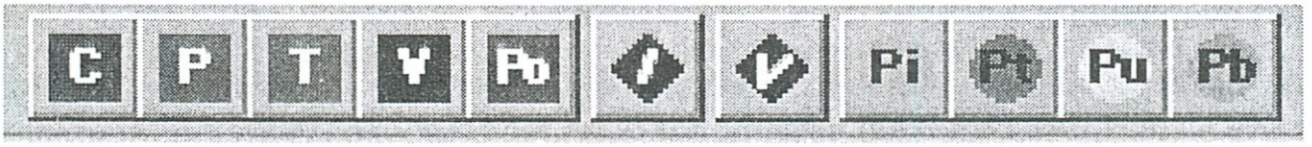
Az osztályok egymáshoz való viszonyán kívül, az osztályokon belüli elemeket, sőt azok minden részletét is vizsgálhatjuk. Láthatjuk azt is, hogy a forráskódban hivatkozunk-e az adott elemre. Ha használunk VCL forrást, akkor annak elemeire való hivatkozásokat is nyomon követhetjük, illetve könnyedén elérhetünk minden metódust, tulajdonságot vagy osztályt a forráskódban. Ha pedig egy hivatkozást választunk ki, akkor a Browser megnyitja a megfelelő forráskódot vagy fájlt, s a kurzort arra a pozícióra állítja, ahol az először előfordul.

Ezt az eszközt más módokon is lehet használni. A Browser SpeedMenu további négy lehetőséget kínál:

- Az Objects parancs az osztályok hierarchiáját mutatja a 26.13. ábrához hasonló módon.
- Az Units parancs az aktuális projektben használt egységeket listázza ki.
- A Globals parancs a globális szimbólumokat jeleníti meg.
- A Symbol paranccsal a forráskódban kereshetünk meg szimbólumokat.



26.13. ábra *Az Object Browser*



26.14. ábra *Az Object Browser gombjai*

Például vehetjük a Windows egységet (ami több, mint 700 kilobájtos forráskóddal rendelkezik) és vehetjük, hogy mely Windows API hívásokat használja, illetve mely konstansok és milyen tulajdonságokkal fordulnak elő benne.

A Browser segítségével a legösszetettebb program is könnyedén elemezhető, hiszen a szerkesztőből könnyedén tudunk a Browserbe és vissza váltani. Ha pedig találnánk egy olyan szimbólumot, amelyről nem tudjuk, hogy mire szolgál, akkor a SpeedMenu **Browse Symbol at Cursor** parancsával a szimbólum definíciójához ugorhatunk. Itt láthatjuk, hogy mi a szimbólum maga és hol kerül felhasználásra. VCL forrás felhasználása esetén a forráskódba is beugorhatunk.

Az Object Browser programmal csak olyan programokat elemezhetünk, amelyeket korábban sikerrel lefordítottunk. Ez azt jelenti, hogy ha egy forráskódot megváltoztatunk s közben egy hibát vétünk, s emiatt a programot nem

lehet lefordítani, akkor nem használhatjuk az Object Browsert arra, hogy megértsük, mi ment rosszul. Ennek elkerülése érdekében az egyes fordítási alkalmak között lehetőségünk van arra, hogy a Browser szimbólumait elmentsük. Ha ezt szeretnénk tenni, akkor válasszuk a **Desktop and Symbols** rádiógombot az Environment Options párbeszédablak **Preferences** oldalán.

Ha ezután a programot bármikor lefordítjuk, akkor a Browser által megkövetelt elemi információk egy DSM kiterjesztésű fájlba kerülnek. Ezzel a fájllal bármikor használhatjuk az Object Browsert, még akkor is, ha a program fordítása nem lehetséges, ugyanis ilyenkor a Browser a megfelelő információkat a fájlból olvassa ki, s nem a programból nyeri ki. A DSM fájlok használatának egy nagy hátránya van, mégpedig az, hogy nagyon nagyok. Ezért lehetőleg csak akkor használjuk őket, amikor igazán szükség van rájuk, azaz akkor, amikor egy nagyon összetett programot fejlesztünk.

Az üzenetvezérlés elemzése

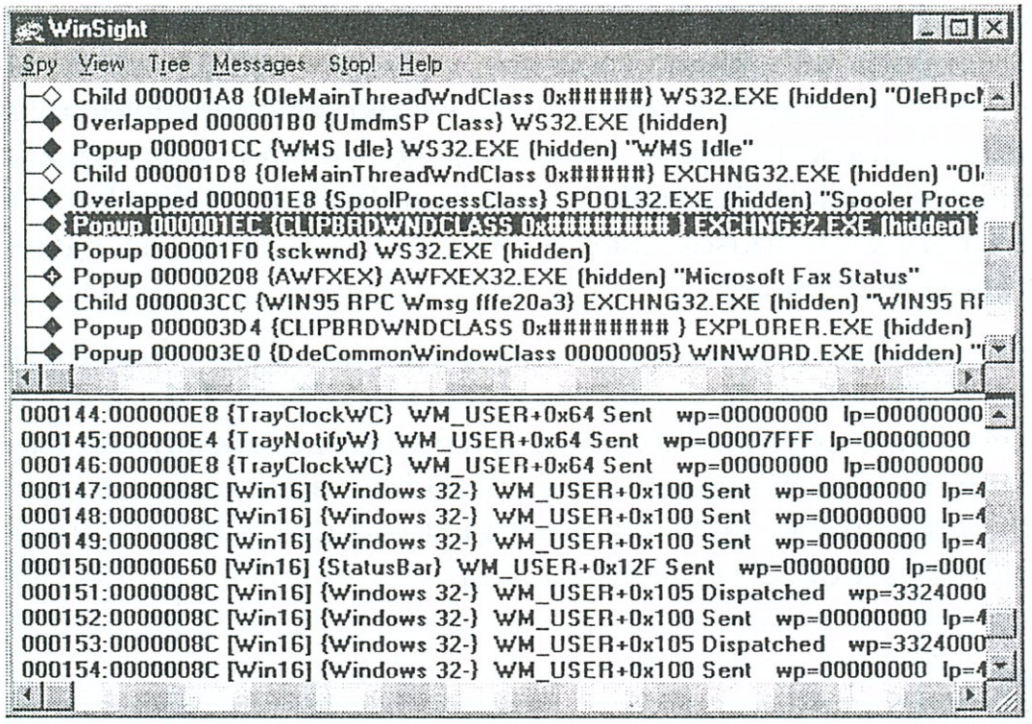
Az előbb tárgyalt két eszköz, a beépített hibakereső és az Object Browser segítségével a program minden része kielemezhető. Azonban vannak olyan esetek, amikor ez mégsem elegendő. A Windowsban időnként arra is szükség lehet, hogy megértsük, miként tartja a kapcsolatot az általunk írt alkalmazás a környezetével. Ilyen esetekben egyes Windows programok rengeteg segítséget nyújthatnak.

Az egyik közülük a WinSight, ami nem más, mint egy többfunkciós Delphi eszköz. Más programokkal a memóriát lehet feltérképezni. A Delphi nem tartalmaz ilyen típusú programot, de szinte nem is kell, mert számos található belőlük könyvekben, CD-ken és egyéb helyeken.

A WinSight használata

A megelőző fejezetekben láthattuk, hogy a WinSight segítségével hogyan lehet egyes alkalmazások működését kielemezni. Itt, ismét felsoroljuk a WinSight néhány alapvető tulajdonságát, néhány új funkcióval és tippel kiegészítve. A WinSight képes arra, hogy a programunk által megjelenített ablakokat hierarchikus fába rendezze, illetve nyomon kövesse, és információkat adjon az üzenetvezérlés pillanatnyi állapotáról.

Mindkét üzemmód elindítható a WinSight **View** menüjéből érhető el: a **Window Tree** és a **Message Trace** parancsok segítségével (a harmadik, a Class View, a WinSight 16 bites verziójának egyik jellegzetes parancsa volt, a 32 bites verzióban nem engedélyezett a használata). Mindkét nézet lehet egyszerre aktív, amit a 26.15. ábrán láthatunk is.



26.15. ábra *A két WinSight panel: az ablakok hierarchikus fája, és a rendszerüzeneteket tartalmazó ablak*

Ha szeretnénk megérteni, hogy mit jelent az összes információ, amit a WinSight szállít nekünk, akkor nem árt, ha alaposan elmélyülünk a Windows, és a Windows alkalmazások felépítésének rejtelmeiben. Bár ez nem képezi a könyv tárgyát, mégis tegyünk egy rövidke kitérőt ezzel kapcsolatban.

A Window Tree nézet: szülőablakok és gyerekablakok

A WinSight Window Tree nézetében az ablakok hierarchikus fája jelenik meg. Ez pontosan, félre nem érthető módon mutatja meg az ablakok közötti összefüggést, amely nagyon fontos mind a Windowsban, mind a Delphiben (lásd még Parent és Controls tulajdonságokat). A WinSight egyik fontos jellemzője, hogy a rejtett és nem látható ablakokat is megmutatja (például az olyan ablakokat is, amelyeket egy pixel nagyságra kicsinyítettünk le). Ugyancsak megjelennek azok az ablakok a fában, amelyeket más ablakok takarnak.

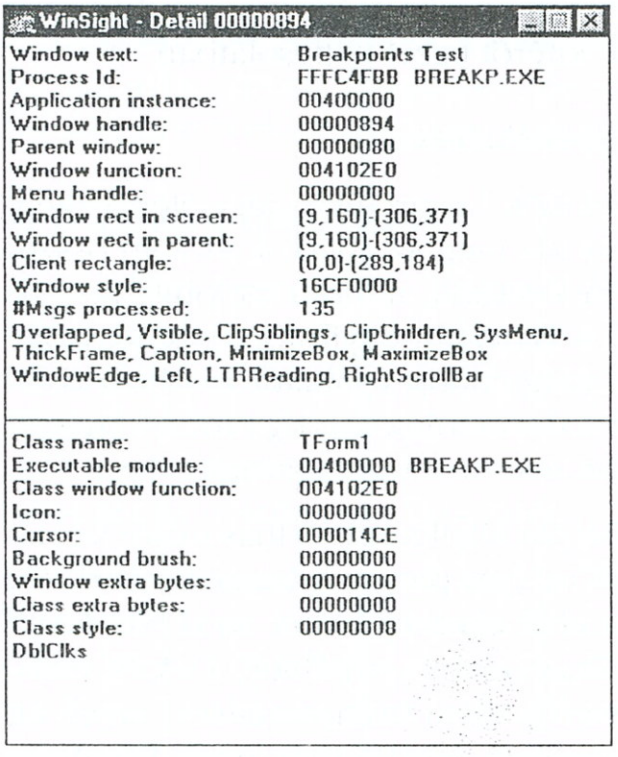
Természetesen, bármely Windows vagy Delphi alkalmazás felhasználói felületének szerkezetéről képet nyerhetünk a WinSight program használatával. Ha például egy általunk írt Delphi programot elemzünk ki, akkor az abban használt formok és vezérlők is megjelennek. Ezzel ellenben néhány Delphi komponens nem jelenik meg. Ezek főleg olyan grafikai komponensek, amelyek nem hoznak létre új ablakot. Az „egy látható elem, egy látható ablak" Windows szabály nem igaz a Delphi programok túlnyomó részére, ami nem is egy olyan rossz dolog, ugyanis az „ablaktalan" grafikus komponensek általában gyorsabbak és kevesebb rendszererőforrást foglalnak le.

A hierarchikus fa minden pontján a WinSight először az ablak típusát jeleníti meg: átlapolt (overlapped), felbukkanó (pop-up), vagy gyermek (child). Általánosságban véve, a leszármazott ablakok a szülőablakokban jönnek létre, míg a takart és az előugró ablakok – akár rendelkeznek szülővel, akár nem – mindig a Desktop teljes felületét használják, és a koordinátáik a teljes képernyőfelületre vonatkoznak. A típusok meghatározzák azt is, hogy az egyes ablaktípusok miként viselkednek.

Az ablak típusa után az ablak leírója, egy hexadecimális szám, következik. Ezt az értéket az ablak létrehozásakor az operációs rendszer dönti el. Ez a szám egyedi, mivel ez azonosítja az ablakot a rendszeren futó összes alkalmazás számára. Mivel ez szinte minden Windows API funkció hívásához elengedhetetlen, ezért a Delphiben elérhető, mint minden `TWinControl` a Handle tulajdonsága, futásidejű és csak olvasható jellemzőkkel.

Megjegyzés

Nem minden Delphi komponens leírója hivatkozik egy ablakra, ugyanis vannak ikon, bitkép és menü leírók is. Ugyancsak nem mondható el minden leíróról, hogy csak olvasható.



26.16. ábra *A WinSight ablaka, amely részletes információkat tartalmaz egy ablakról és annak osztályáról, jelen esetben egy Delphi formról*

Az ez után következő információk: az osztály neve, a futtatható állomány neve, az ablak helye és nagysága, címe vagy címkéje. Az ablakokról a 26.16. ábrához hasonló részletesebb információs panelt is kaphatunk, amely a fentieken kívül tartalmazza annak az alkalmazásnak a leíróját is, amely az ablakot létrehozta (ezt egyébként a 25. fejezetben megismert HIstance globális változó tartalmazza), a menü leíróját, az ablakhoz kapcsolódó eljárás memóriacímét, a kliens négyzetét, és az ablak stílusához kapcsolódó jelzők nevét és állapotát.

A felsorolás végén kaptak helyet az ablak osztályával kapcsolatos információk. Ezek nem Delphi, hanem Windows osztályokra való hivatkozások. Annak érdekében, hogy megérthessük, hogy mi is az az ablak osztály, először át kell néznünk, hogy mi is történik akkor, amikor a rendszer egy ablakot hoz létre, és hogyan válaszol azok üzeneteire. Röviden, tetszőleges ablak létrehozásához (legyen az a főképernyő vagy egy kis vezérlő ablaka) a Windows környezete megköveteli, hogy megadjuk az ablakhoz tartozó osztályt, s hogy formálisan bejegyezzük azt az osztályt. Ennek érdekében a legtöbb C program egy ablak létrehozása előtt mindig bejegyzi az ahhoz tartozó osztályt, hogy az létezzen az ablak létrehozásának pillanatában. A rendszerben szereplő minden egyes ablaknak van osztálya, más néven WNDCLASS, amelyet a regisztrálás közben használt adatstruktúra határoz meg (a Delphiben ez a `TWndClass`).

A Windowsban használt osztály kifejezésnek nagyon kevés köze van ahhoz az osztályhoz, amelyet a fejlett, objektum-orientált programozási rendszerek használnak. A Windows WNDCLSS adatstruktúrájának előre meghatározott számú és típusú mezője van, az egyik ezek közül az, amely az eljárás címét tartalmazza (más néven ablakeljárás vagy WndProc). Ez az eljárás az, amely válaszol a rendszer által az osztályba tartozó ablakok számára küldött üzenetre. Ebből kifolyólag az átlagos ablakeljárás egy hatalmas `case` kifejezéssel rendelkezik, amelybe az összes adható válasz „bele van drótozva" (ebből is látszik, hogy a Delphi és más C++ fejlesztőkörnyezetek ablakkezelése sokkal átgondoltabb és jobb kivitelezésű, mint a Windows megközelítése).

A WinSight által megjelenített ablakok osztályinformációi tartalmazzák az osztály nevét, az azt létrehozó futtatható állomány vagy dinamikusan kapcsolt könyvtár (DLL) nevét, az osztályhoz tartozó ablakeljárás címét és néhány stílusjelzőt.

Ha tapasztalt Windows programozók vagyunk, akkor észrevehetjük, hogy az osztály és az ablak ablakeljárása nem egyezik meg. Alapállapotban minden ablak az osztályához rendelt ablakeljárást használja mindaddig, amíg ezt valami meg nem változtatja egy subclassing nevű technikával (ennek semmi köze nincsen az Object Pascal hasonló kifejezéséhez). Az alosztályosítás (subclassing) főleg VCL-ben használatos egy ablak-objektum viselkedésének leírókon keresztüli beállítására. A Delphi elemekben található Windows eljárások annyira el vannak rejtve, hogy szinte lehetetlen felfedezni őket. A

Windows eljárás meghívja az ablak ablakeljárását, amelyen keresztül már közvetlenül elérjük a megadott objektumot. Ezzel ellentétben, a hagyományos Windows program egy közös ablakeljárást használ minden egyes ablakhoz. A Delphi megoldása ezért sokkal nagyobb rugalmasságot biztosít.

Üzenetvezérlés és a Delphi események

A WinSight használatán keresztül megtanulhatjuk, hogy hogyan lehet a Delphin belüli üzenetváltásokat nyomon követni. Annak érdekében, hogy professzionális Delphi programozóvá váljunk, meg kell ismerkednünk azzal, hogy mi is történik akkor, ha a felhasználó leüt egy billentyűt. Mint minden Windows alkalmazás, a Delphi programok is eseményvezéreltek, azaz a program végrehajtása egy esemény hatására következik be. A Delphi események mögött Windows üzenetek rejtőznek, még akkor is, ha nem létesíthető egy az egyhez megfeleltetés közöttük.

Tipp

A programozási zsargonban a spy (kém) szó használatos az üzenetek nyomkövetésére, mert az eredeti programnak, amely lehetővé tette az ilyen fajta tényked ést, Spy volt a neve.

A Windowsban sokkal több üzenet van, mint amennyi Delphi esemény létezik, de egyes Delphi események magasabb szinten képződnek, mint a Windows üzenetek. Például a Windows csak nagyon kevés támogatást ad az objektumok egérrel való vonszolásához, míg a Delphi komponensek mindegyike teljes körű drag-and-drop támogatást élvez. Természetesen a WinSight semmit sem tud a Delphi eseményeiről, ezért nekünk kell kitalálni a megfelelő összefüggéseket az események és a rendszerüzenetek között (vagy tanulmányozzuk a VCL forráskódot, ha rendelkezésünkre áll). A WinSight olvasható formában megmutatja, hogy milyen rendszerüzenetek érték el az ablakot, beleértve, hogy mely ablak kapta őket, milyen paraméterrel, illetve mi a neve vagy milyen osztályba tartozik. A WinSight **Message** menüjének **Options** parancsával kiválaszthatjuk, hogy csak azok az üzenetek jelenjenek meg, amelyek érdekelnek minket.

Általában véve a Delphi programozók számára a rendszerüzenetek nyomon követése akkor a legkívánatosabb, amikor a program ablakkal kapcsolatos műveletekkel kapcsolatban rendetlenkedik. A rendszerüzenetek alapján a legtöbb esetben szinte azonnal láthatjuk, hogy mi is ment rosszul. Ugyancsak

ehhez a módszerhez kell folyamodnunk, ha a Windows belső rendszerüzeneteit közvetlenül, az eseménykezelők nélkül, használjuk fel. A WinSight használatával több információt tudunk a rendszerüzenetről és annak paramétereiről.

Az elküldött üzenetek nyomon követése

A rendszerüzenetek nyomon követését úgy is megoldhatjuk, hogy azokat egyes Delphi alkalmazásokon belül „füleljük le". Ha csak a kifüggesztett rendszerüzenetekre (a PostMessage által elküldött) koncentrálunk, azaz kihagyjuk az elküldött üzeneteket (amit a SendMessage hoz létre), akkor csak a `TApplication` osztály egyetlen eseményét kísérjük figyelemmel, az OnMessage-et. Ez utóbbinak a szerepe az, hogy lehetőséget biztosítson az egyes alkalmazások számára, hogy megszűrjék a bejövő üzeneteket, illetve, hogy egyes üzeneteket speciális formában küldhessenek el. Ezzel például elküldhetünk egy üzenetet az Application objektumhoz kapcsolódó ablakhoz, amelynek egyébként nincsen specifikus esemény eseménykezelője (ezzel már foglalkoztunk a 25. fejezetben).

Az MsgFlow példaprogramban, ennek ellenére, az alkalmazás által kapott rendszerüzeneteket vesszük szemügyre (azaz a kifüggesztett üzeneteket). Az összes beérkezett üzenetről egy rövid leírás jelenik meg egy, a formot teljesen letakaró listapanelen. Itt lehetőleg a Courier betűkészletet használjuk, mert akkor kapunk szép, egymás alá rendezett külalakot.

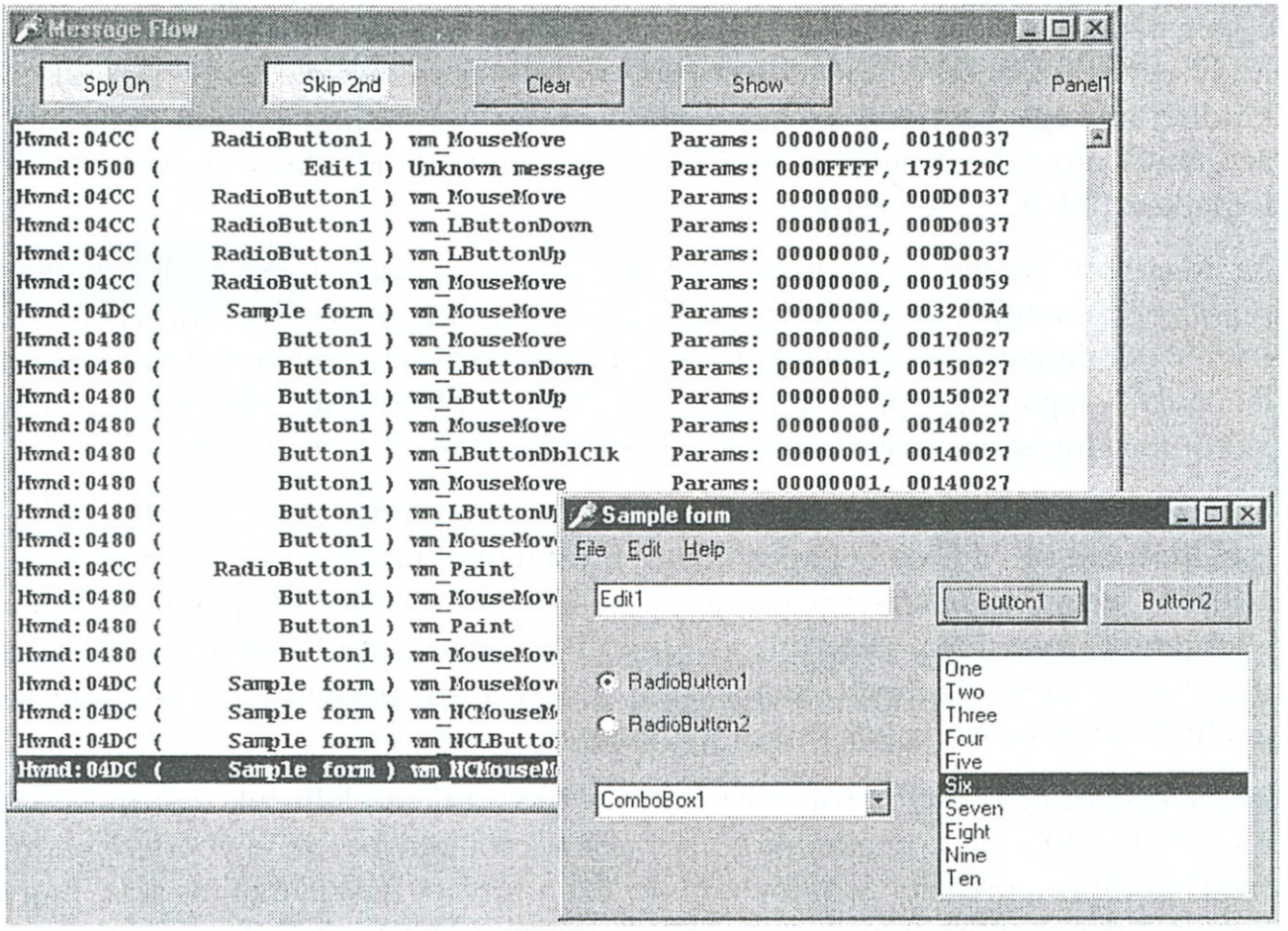
Az eszköztáron található gombok segítségével a rendszerüzenetek figyelemmel kisérése ki-, illetve bekapcsolható, a listapanel taralma töröllhető és megadható, hogy a többszörös, egymást követő üzeneteket csak egyszer jelenjenek meg. Ha például az egeret megmozgatjuk, akkor rengeteg `wm_MouseMove` üzenetet kaphatunk, amelyet nyugodtan el lehet hagyni anélkül, hogy hasznos információkat vesztenénk el.

Hogy igazán érezzük, miről is van szó, a program rendelkezik egy második formmal is, (amely a negyedik gombbal kapcsolható be) amelyben véletlenszerű komponensekkel találkozhatunk. Ezt a formot használhatjuk arra, hogy lássuk, milyen rendszerüzenetek áramlanak egy általános Delphi ablak felé, illetve felől (26.17. ábra).

A program alapötlete elég egyszerű. Hozzunk létre egy eseménykezelőt az alkalmazás OnMessage eseménye számára, majd telepítsük ezt a FormCreate metódusba:

```
type
  TForm1 = class(TForm)
    ...
  private
    Skipping, Spying: Boolean;
    LastMessage: TMsg;
```

```
    public
      procedure HandleMessage (var Msg: TMsg;
        var Handled: Boolean);
    end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := HandleMessage;
  Skipping := False;
  Spying := False;
end;
```



26.17. ábra *A MsgForm program futás közben, a második form másolatával*

Vegyük észre, hogy néhány logikai jelző meghatározása és kezdeti értéke az eszköztár gombjainak állapotától függ. Ezektől a jelzőktől és a legutolsó üzenet értékétől függ (a TMsg szerkezetével lentebb foglalkozunk), hogy amikor egy üzenet megérkezik az alkalmazáshoz, az megjelenik-e a képernyőn. A HandleMessage programmagja csak akkor hajtódik végre, ha a következő három feltétel teljesül:

- A Spying logikai jelző értéke igaz. Ezt a felhasználó ezt a Spy On gombra való kattintással egyszerűen megváltoztathatja.
- Az üzenet nem a listapanelhez tartozik. Ezzel megakadályozhatjuk az önmagát hívó rendszerüzeneteket.
- Ha a Skipping jelző engedélyezve van (ehhez kattintsunk az eszköztáron a második gombra), akkor az üzenet csak akkor jelenik meg, ha a címzettje, vagy az üzenet tartalma az előbbitől eltér. Többszörös hívások esetén, mint például az egér, csak az első sor jelenik meg.

Ezeket a feltételeket a következő összetett `if` kifejezéssel ellenőrizhetjük:

```
if Spying and (Msg.Hwnd <> Listbox1.Handle) and not
  (Skipping and (LastMessage.Message = Msg.Message) and
  (LastMessage.Hwnd = Msg.Hwnd)) then
  ...
```

A HandleMessage forráskódjában minden egyes üzenethez egy-egy sor szöveget rendelhetünk, ezzel is valamilyen értelmes tartalommal tölthetjük meg. Az OnEvent esemény eseménykezelőjének paramétere TMsg típusú, ami nem más, mint alapvető információk gyűjteménye, úgy, mint a címzett, az üzenet száma és néhány más érték. Ez az az információ, amit a Windows az alkalmazás felé küld. A közvetlen üzenetolvasás helyett a következő két módszer egyikét válasszuk.

Először, az MsgFlow program az ablak címkéjét a leírójával együtt jeleníti meg, amelyet néhány Windows API eljáráson keresztül ér el. Ezek közül egy a GetWindowText, amely minden fajta ablakkal együttműködik. A szöveg először 15 karakter hosszúra lesz formázva a Delphi Format funkciója által, majd a Line karakterlánchoz lesz fűzve, amelyet majd a metódus végén helyezünk el a listapanel végére. Ha a kapott szöveg véletlenül üres (az ablaknak nincsen címkéje), akkor az Unknown szöveget fogjuk hozzáfőzni a Line karakterlánchoz. Íme a metódus első részének programrészlete:

```
// output the hex value of the handle
        Line := 'Hwnd:' + IntToHex (Msg.Hwnd, 4);
        {get the caption from the handle,
        using an API function}
        GetWindowText (Msg.Hwnd, PChar (Caption),
          Length (Caption));
        Caption := PChar (Caption); // recast
        if Caption = '' then
          Caption := 'Unknown';
        // format the caption in 15 characters
        AppendStr (Line, Format (' ( %15s ) ',
            [Caption]));
```

Figyeljük meg, hogy a stringet PChar-rá, majd ismét stringgé kell alakítani, ugyanis csak így lehet a Delphit rávenni arra, hogy ismét számolja ki a karakterlánc hosszát (bővebben a 4. fejezetben).

A második előrelépés az, hogy az üzenetek kódja helyett azok neve jelenik meg. Ez nem olyan egyszerű, mint amilyennek látszik, mivel az üzenetek nevei nem igazi stringek, hanem konstansok nevei, amelyeket a MESSAGES.PAS rendszerfájl ad meg. Íme pár sor ebből a fájlból:

```
const
  wm_Null          = $0000;
  wm_Create        = $0001;
  wm_Destroy       = $0002;
  wm_Move          = $0003;
```

Ez az információ fontos, de nem használható az előbbi példánkban. Ennek ellenére felhasználhatjuk néhány esetben.

A probléma az, hogy nem hozhatunk létre belőle tömböt, hiszen nem minden értékhez tartozik rendszerüzenet. Emlékezhetünk arra is, hogy a `TStringList` osztály rendelkezik stringekkel és objektumokkal is. Egy objektum `TStringList`-ben való tárolása négy bájt tárolását jelenti, s ezt ebben az esetben ki is használhatjuk, azaz az objektum helyett az üzenet számát adjuk meg. Ezt teszi a következő programrészlet is:

```
var
  MsgList: TStringList;
...
  MsgList := TStringList.Create;
  MsgList.AddObject ('wm_Null        ', TObject($0000));
  MsgList.AddObject ('wm_Create      ', TObject($0001));
  MsgList.AddObject ('wm_Destroy     ', TObject($0002));
  MsgList.AddObject ('wm_Move        ', TObject($0003));
```

Ez a programrészletet (mellesleg jó pár oldal hosszú) úgy hoztuk létre, hogy kicseréltük a `wm` szöveget a `MsgList.AddObject(`wm szöveggel, az egyenlőségjelet a típus-átalakítás számával, és két zárójelet tettünk a sor végére. A számból `TObject`-be való átalakítás egy rendszerszintű programozói fogás, amelyet általában kevesen szeretnek, de ebben az esetben nem volt egyszerűbb megoldás.

Ez a programrészlet egy külön egységben kapott helyet, amely a MLIST.PAS nevet kapta. Ennek az egységnek az interfésze tartalmaz egy olyan funkciót, amely az adott hibakódhoz tartozó hibaüzenetet adja vissza:

```
function GetMessageName (Msg: Integer): string;
var
  N: Integer;
begin
```

```
    N := MsgList.IndexOfObject (TObject(Msg));
    if N >= 0 then
      Result := MsgList.Strings [N]
    else if Msg >= wm_User then
      Result := 'wm_User message      '
    else
      Result := 'Unknown message      ';
  end;
```

Ez a szöveg építi fel a karakterláncot az üzenet leírásával és számával együtt. Íme az `if` kifejezés második fele (az elsőt pár bekezdéssel feljebb lehet találni):

```
  AppendStr (Line, GetMessageName (Msg.Message));
  {add the hexadecimal output of the two message
   parameters}
  AppendStr (Line, 'Params: ' + IntToHex (
    Msg.wParam, 8) + ', ' + IntToHex (Msg.lParam, 8));
  {add the line, selecting it}
  ListBox1.ItemIndex := ListBox1.Items.Add (Line);
  {store the message, to compare it with the next one}
  LastMessage := Msg;
```

Végül, a karakterlánc a listához csatolódik. Az Add metódus visszatérési értéke használatos a listapanel új elemének kijelölésére. Ekkor az aktuális üzenet szerkezete mentésre kerül, s ezután már összehasonlíthatjuk a soron következő üzenettel, s azt akár el is dobhatjuk, ha arra támad kedvünk.

Ha most a programot futtatjuk, akkor a kapott üzeneteket egy része jelenik csak meg, hiszen csak a kifüggesztett üzenetekre vagyunk kíváncsiak. A legtöbb üzenet valószínűleg az egér mozgatásával lesz kapcsolatban. Figyeljük meg, hogy láthatjuk az ablak címkéjét. Ezt azért nem szüntettük meg, hogy az ablakhoz és a gombokhoz kapcsolódó üzenetek értelmezhetőségét nagyban növeljük. Ez utóbbiak nem ablakozott üzenetek, ezért nem fogadhatnak üzeneteket a rendszertől. Egy gombon való kattintás nem jelent mást, mint a panelen való kattintást.

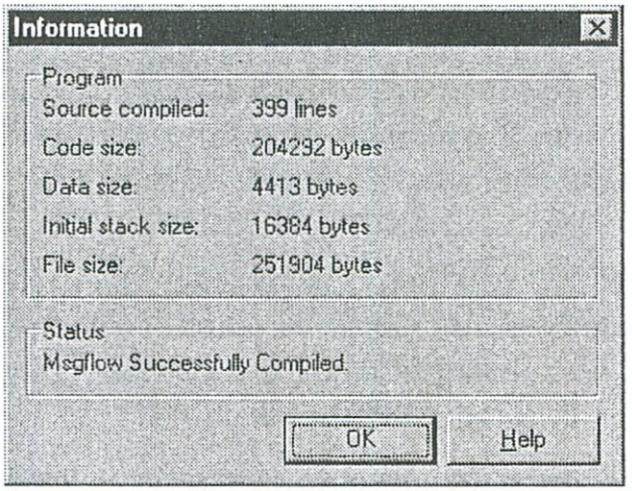
Az alkalmazás a memóriában

Mind a Windows 95, mind a Windows NT operációs rendszerben minden alkalmazás egy összefüggő, 4GB-os címtartományt lát. Igazság szerint, a Win95 és az NT az alkalmazás számára csak azon memóriacímek használatát engedélyezi, míg más memória más részében az alkalmazások által elérhető DLL-ek találhatóak meg.

Minden alkalmazás hasonló szerkezettel rendelkezik, de nem férhet más alkalmazások által lefoglalt memóriához. Minden alkalmazás, természetes, elérheti az a DLL könyvtárak kódját (az adatait nem!). A Win32 rendszerekben

a futtatható állományok, köztük a DLL-ek is, memória-térképezett memóriába kerülnek, így minden alkalmazás számára láthatóak. Azaz, ha két alkalmazás ugyanazt a programrészletet alkalmazza, ezért nincsen szükség arra, hogy kétszer betöltsük a memóriába ugyanazt.

Ahhoz, hogy megkapjuk az alkalmazás által lefoglalt memóriarekeszek nagyságát a Delphi Compiler menüjéből tudhatjuk meg. Az Information párbeszédablakban (26.18. ábra) megjelenik a lefordított program és a globális változók nagysága, az alkalmazás vermének mérete, amely a projekt beállításaitól függ. Win95 és NT alatt szinte nincsenek határai annak, hogy egy program mennyi memóriát használ. Az egyetlen feltétel az, hogy legyen elég fizikai memóriának, és elég hely a merevlemezen. Ezért hozhatunk létre akkora tömböket és karakterláncokat, amekkorákat csak akarunk.



26.18. ábra *A MsgForm példaprogram fordítási információi*

A lefordított program nagyságán kívül a futtatható állomány tartalmaz néhány forrásmeghatározást is, úgy, mint ikonok, kurzorok, karakterlánc táblák, és így tovább. Továbbá, különböző eszközök és erőforrások csak akkor kerülnek be a memóriába, ha éppen szükségük van rá, illetve automatikusan kikerülnek onnan, ha tovább már nincsen szükség rájuk

A programkódon és az erőforrásokon kívül, amelyek a futtatható állományokból kerülnek a memóriába, a memória tartalmazza az alkalmazások számára fenntartott vermeket, globális változókat és a dinamikusan bővülő memóriát. A programunk által használt globális változók nagysága attól függ, hogy hány globális változót hoztunk létre a programunkban, illetve a VCL hányat ad meg. Egy kisebb projekt csak néhány kilobájtot foglal le erre a célra. A Delphiben a legelfogadottabb módszer a program globális változói által lefoglalt memória nagyságának meghatározására az, hogy a Compiler Information párbeszédablakon keresztül megnézzük.

A verem eredeti nagyságát a **Linker** projekt opciókon keresztül lehet szabályozni, s akár a maximumát is meg lehet adni (ez általában egy nagyon nagy szám). A verem gyakorlatilag az eljárások, funkciók és metódusok hívásánál használt tároló, a lokális változók, paraméterek és visszatérési értékek átadására szolgál. Ennek ellenére a Delphi alapértelmezett hívási eljárása a regiszterhívás, amely a processzor regisztereit használja az értékek átadására. Még a DLL-ek funkciói, beleértve a rendszerkönyvtárakat is, alkalmaznak egy kevés veremmemóriát az adatok kicserélésére, átadására.

A fennmaradó memória, a heap, az összes, dinamikusan kiosztott memóriát és a Delphi összes objektumát tartalmazza. A Windows 3.1 alatt kétféle heap memóriát használhatunk. A kis méretű, hatékony lokális heapet, illetve a nagy, nem annyira hatékony globális heapet. A Win95 és az NT alatt minden alkalmazás tetszőleges méretű heappel rendelkezhet. Ahogyan arról már szót ejtettünk, a Windows 95-ben és az NT-ben megszűnt a globális heap használata.

Tipp

Egy futtatható állomány belső felépítését többféleképpen is kielemezhetjük. Windows 95 alatt az Intéző helyi menüjéből a Nézőke paranccsal érhetjük el ezt a szolgáltatást. A Delphiben hasonló funkciót tölt be a parancssorból indítható TDUMP.EXE, amellyel ezeken felül, többek között, DLL-ek vizsgálatára is alkalmazhatunk. Más programok is használhatóak e célra, de azokról bővebben csak a következő fejezetben fogunk beszélni.

A Windows rendszermemóriája

Az alkalmazások a programkódjuk számára lefoglalt memórián kívül egy kevés rendszermemóriát is lefoglalnak. Például a Windows rendszerkönyvtárak saját memóriaszegmenssel rendelkeznek. Ezeknek a szegmenseknek a maximális nagysága 64 kilobájtra korlátozódik a Windows 3.1-es verziójánál, illetve a Windows 95 16 bites DLL-jeinél. A Windowsban három fő rendszerkönyvtár található: Kernel, User és GDI.

A Kernel DLL feladata a modulok és processzek számára memóriablokkok lefoglalása és karbantartása. A modulokhoz hozzátartozik a rendszerbe töltött összes futtatható állomány, beleértve a programokat, dinamikus könyvtárakat, rendszermeghajtó eszközöket, és a többi végrehajtható fájlt.

A User DLL az ablakok és menük heap memóriáját hozza létre és kezeli. Minden egyes ablaknak és ablakosztálynak megvan a saját heapje. A Windows 3.1-ben ez okozta a legtöbb fejfájást, mert 700 ablak létrehozása gyakorlatilag elfogyasztotta a heapet, és ez lefagyaszthatta a rendszer. A Windows 95-ben ez a határ magasabb, 16000 körül mozog.

Szerencsére a Delphiben számos módszert alkalmazhatunk arra, hogy az ablakok számát csökkenthessük. Ahogyan azt már korábban láthattunk, nem minden Delphi komponensnek van szüksége ablakra. Az „ablaktalan" komponensek használatával megtakaríthatunk egy kevés memóriát a User heapben, ilyenek például a SpeedButton, az Image, a Label komponensek, de még nagyon sok más is létezik. Mivel az összes „ablaktalan" komponens a `TGraphicsControl` alosztályaiban található, így könnyedén megkereshetjük őket a VCL komponensek hierarchiájában.

A User DLL-nek van egy nagy előnye. Ha egy általunk indított alkalmazás lefut, akkor kilépésekor a heapje automatikusan megszűnik, felszabadítva a memória egy részét. Sajnos nem ez a helyzet a GDI heappel.

A GDI DLL a grafikus eszközökre – tollak, ecsetek, fontok, bitmap – vonatkozó információt tárol. Ahányszor létrehozunk egy ilyen eszközt, mi felelünk azért, hogy megfelelően kitakarítsuk a memóriából. Ha ezt nem tesszük, akkor hiába lép ki a programunk, az eszköz által lefoglalt memória nem szabadul fel. Ebből kifolyólag, bár a GDI objektumok sokkal kisebbek, mint az ablakszerkezetek a Windowsban, egy rosszul megírt program, ami GDI objektumokat hoz létre, de elfelejt megszüntetni, komoly bajokat okozhat.

A szabad rendszermemória

A GDI és User heapet sokszor rendszererőforrásnak is nevezik. Ez nem egy szerencsés elnevezés, mivel az erőforrás kifejezést a programba beágyazott grafikus elemekre is szokás alkalmazni, ahogyan azt a megelőző fejezetekben is láthattuk. A két erőforrásnak semmi köze nincsen egymáshoz.

A legtöbb Windows-t ismerő felhasználó tisztában van a heapre vonatkozó erőforrások jelentésével, hiszen a legtöbb program About párbeszédablaka megmutatja a rendelkezésre álló rendszer erőforrás mértékét, százalékban. Itt bizonyosodhatunk meg arról, hogy nem futunk ki a rendelkezésre álló erőforrásokból, bár ez a Windows 95 esetén ritkán fordul elő.

Mivel a Win16-os funkciók a Windows 95 alatt csak nagyon nehezen érhetőek el, ezért csak a globális memóriaadatok lekérdezésével valósítható meg a rendszererőforrások állapotának nyomon követése.

A rendszermemória állapotát a GlobalMemoryStatus API funkción keresztül tudhatjuk meg. Az ennek a bemutatására íródott Mem példaprogram két körábraként konfigurált (azaz a típusa `gkPie`) csúszkát, az azokhoz tartozó

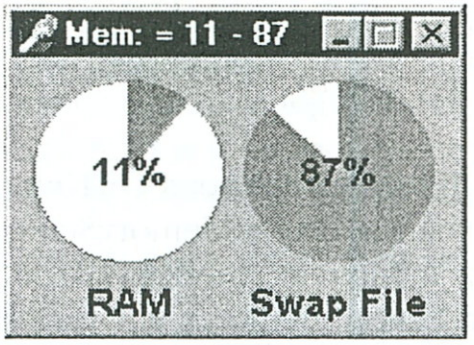
címkét és egy időzítőt tartalmaz. Egyszerűbb, ha egy időzítő segítségével frissítjük a formot, mintha a felhasználót utasítanánk arra, hogy időnként kattintson egy gombra vagy menüparancsra.

Az egyetlen valódi metódusa ennek a formnak a Timer komponens OnTimer eseményének eseménykezelője. Ebben az eljárásban a GlobalMemoryStatus által visszaadott információkat dolgozzuk fel. Ez a funkció, többek között, visszatér a szabad fizikai memória nagyságával, a fizikai memória, a virtuális memória és a szabad virtuális memória nagyságával. Ezek megfelelő elosztásával megkapjuk a szabad rendszermemória százalékos arányát.

```
procedure TMemForm.Timer1Timer(Sender: TObject);
var
  MemInfo : TMemoryStatus;
begin
  MemInfo.dwLength := Sizeof (MemInfo);
  GlobalMemoryStatus (MemInfo);
  RamGauge.Progress := MemInfo.dwAvailPhys div
    (MemInfo.dwTotalPhys div 100);
  VirtualGauge.Progress := MemInfo.dwAvailPageFile div
    (MemInfo.dwTotalPageFile div 100);
  Caption := Format ('Mem: = %d - %d',
    [RamGauge.Progress, VirtualGauge.Progress]);
  Application.Title := Caption;

  {if value is low, turn color to red}
  if (RamGauge.Progress < 5) then
    RamGauge.ForeColor := clRed
  else
    RamGauge.ForeColor := clLime;
  if (VirtualGauge.Progress < 20) then
    VirtualGauge.ForeColor := clRed
  else
    VirtualGauge.ForeColor := clLime;
end;
```

Láthatjuk, hogy az így kapott eredményt az ablak fejlécébe is bemásoljuk, így akkor is láthatjuk, ha a program le lett kicsinyítve. Természetesen sokkal szebb, ha színes körábrákat nézünk a két szám helyett. Utoljára még meg kell említeni, hogy a szabad memóriát jelképező körábra színe változik attól függően, hogy mennyi még a szabad memória. A programról készült kép a 26.19. ábrán látható.

A program szépen néz ki, de sokkal jobb lenne, ha a Windows 95 Tálcáján is megjelenne. Pontosan erről lesz szó a következő fejezetben azután, hogy megtanuljuk, hogyan is használjunk az ikonokat a Windows 95 alatt.



26.19. ábra *Példa a Mem példaprogram futására*

Hogyan tovább?

Ebben a fejezetben olyan módszerekkel ismerkedtünk meg, amelyekkel programokat és azok Windows-zal való viszonyát lehet vizsgálni. Egy Windows alkalmazás nagyon ritkán foglalja le egymaga a teljes rendszert. A legtöbb esetben más alkalmazásokkal osztozik a rendelkezésre álló erőforrásokon, így nem meglepő, hogy az alkalmazás nagyon szoros kötelék fűzi a rendszerhez, és közvetve, más alkalmazásokhoz. A más alkalmazások jelenléte és tulajdonságai nagyban befolyásolják a rendszer teljesítményét és stabilitását.

A Windows 95 és NT alatt kevesebb memóriával kapcsolatos problémával találkozhatunk, mint a 3.1-es verziónál, annak ellenére, hogy néhány 16 bites verzióból öröklött probléma megmaradt (amelyek még mindig a 16 bites DLL-re támaszkodnak). Minden alkalmazásnak megvan a saját számára kiutalt memória, amelybe más alkalmazás nem piszkálhat bele, így a legtöbb veszélyes helyzet kikerülhető.

Egyetlen témával nem foglalkoztunk még, az erőforrások szerepével egy alkalmazáson belül. Nem foglalkoztunk ezzel a 25. fejezetben, amelyben az alkalmazásokról volt szó, sem itt, amelynek a fő témája a memória volt, ez ugyanis a következő fejezet témája lesz.

27. FEJEZET

Erőforrások használata

E fejezet tartalma:

- Windows erőforrások
- Programok és formok ikonjai
- A kurzor használata a Delphiben
- Stringtábla erőforrások használata
- Verziószámok kezelése

Erőforrások használata

Ahogy az előző fejezetben láthattuk, a Windows alatti futtatható fájlok speciális adatrészeket tartalmazhatnak, az erőforrásokat. Ezek között találhatóak grafikus erőforrások, mint amilyen a program ikonja, a használt bittérképek és a kurzorok. Más típusú erőforrások írják le a menük és a párbeszédablakok sablonjait, a stringtáblákat, és még sok egyéb elemet (ne keverjük össze ezt az erőforrás fogalmat a Windows környezetben szintén használt másik jelentésével, ahol a rendszerkönyvtárak által felhasználható szabad memóriaterület méretét adja meg).

A legtöbb Delphi alkalmazás használ erőforrásokat, hiszen legalább egy ikonja minden programnak van. Ahogy már korábbi példaprogramokban is láthattuk, a futtatható fájl erőforrás részében elhelyezhetünk bittérképes ábrákat, tehát nincs szükségünk külön fájlok használatára ezek tárolására. Ebben a fejezetben áttekintjük a Windows erőforrások használatát a Delphiben, és megnézünk néhány érdekes példát.

Windows erőforrások

A 26. fejezetben megbeszéltük az erőforrások szerepét a memóriahasználat szempontjából. Láttuk, hogy az erőforrások a futtatható fájl különálló részeiben kapnak helyet és, hogy csak akkor kerülnek be a memóriába, amikor szükség van rájuk, illetve, ha nincs rájuk szükség, akkor kikerülhetnek onnan. Szintén megbeszéltük, hogy csak olvasható adatterületnek minősülnek. Nem szóltunk még viszont az erőforrások különböző típusairól és egyéni felhasználásukról. A 27.1. táblázat az összes Windows erőforrást bemutatja.

27.1. táblázat

A Windows erőforrástípusai

Erőforrás	Resource	Leírás
Ikonok	Icon	32x32 vagy 16x16 pixeles bittérképek, korlátozott számú szín felhasználásával. Az alkalmazások grafikus azonosítására használható.
Menü sablonok	Menu template	Egy menü felépítését határozzák meg. Delphiben nem használjuk.
Párbeszédablak sablonok	Dialog box template	Egy párbeszédablak felépítését határozzák meg. Delphiben nem használjuk.
Bittérképek	Bitmap	Általános célú bittérképeket definiál. A Delphi nem támogatja ezek közvetlen használatát, de innen is betölthetünk képet, fájlok használata helyett.

27.1. táblázat
A Windows erőforrástípusai (folytatás)

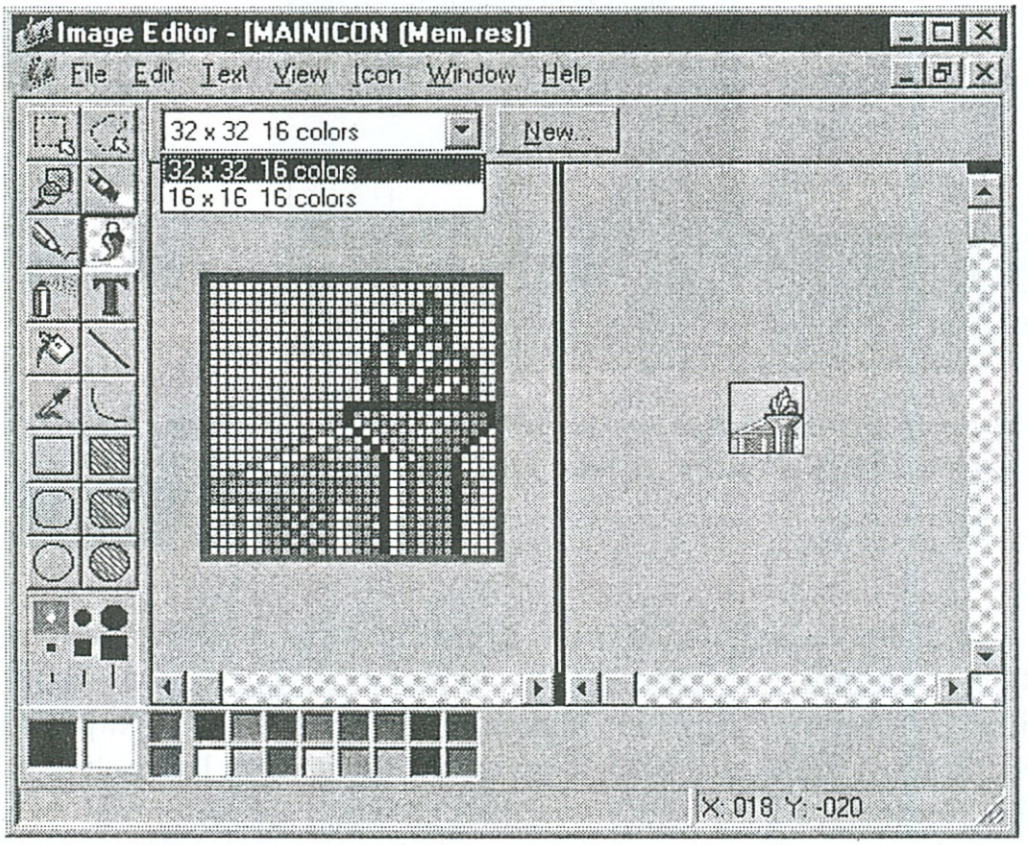
Erőforrás	Resource	Leírás
Betűkészletek	Font	Egyetlen programhoz definiálhatunk vele új betűkészletet. Saját betűkészlet erőforrásokat ritkán használunk Windows alkalmazásokban, mert szebb megoldás, ha a betűkészletet különálló fájlként telepítjük a számítógépre.
Gyorsító táblák	Accelerator table	Menüparancsokhoz rendelt gyorsbillentyűket tartalmaz. Delphiben nem használjuk.
Verziószám	Version Information	Egy speciális erőforrás a program verziószámának és a készítő adatainak tárolására. Ezek megfelelő kezelése a DLL fájloknál különösen fontos, ezért a Delphi közvetlenül támogatja a verziószám kezelését a projekt beállítások segítségével. Ennek részleteit a fejezet végén találhatjuk.
Egyéni erőforrások	Custom Resources	Felhasználó által definiált erőforrások. Bár közvetlenül ritkán használjuk, a Delphi így tárolja a formok bináris leírását.

Mielőtt megnéznénk az erőforrások speciális felhasználási területeit a Delphiben, foglalkozzunk az erőforrások készítéséhez biztosított eszközökkel. A Delphi tartalmazza az Image Editor programot, mellyel ikonokat, kurzorokat és általános bitképeket is szerkeszthetünk. Néhány esetben hasznos lehet egy teljes erőforrás kezelő program, például a Borland Resource Workshop, ami a Delphihez nem tartozik hozzá. Beszerezhetőek shareware erőforrás szerkesztők is.

Az erőforrás szerkesztők használata

A Delphi Image Editorát a Tools menü megfelelő parancsával indíthatjuk el. Ezzel négyféle fájlt készíthetünk. Ezek közül három, speciális erőforrásoknak felelnek meg (ICO, CUR, BMP). A negyedik a lefordított erőforrás fájl (RES), ami mind a fenti három, grafikus erőforrástípust is tartalmazhatja. Saját RES fájlokban szerepelhetnek további, akármilyen típusú erőforrások is. A Delphi programjainkhoz ezeken a RES fájlokon keresztül kapcsolhatjuk az erőforrásokat a $R fordítói direktívával. Ennek módját később részletesen ismertetem.

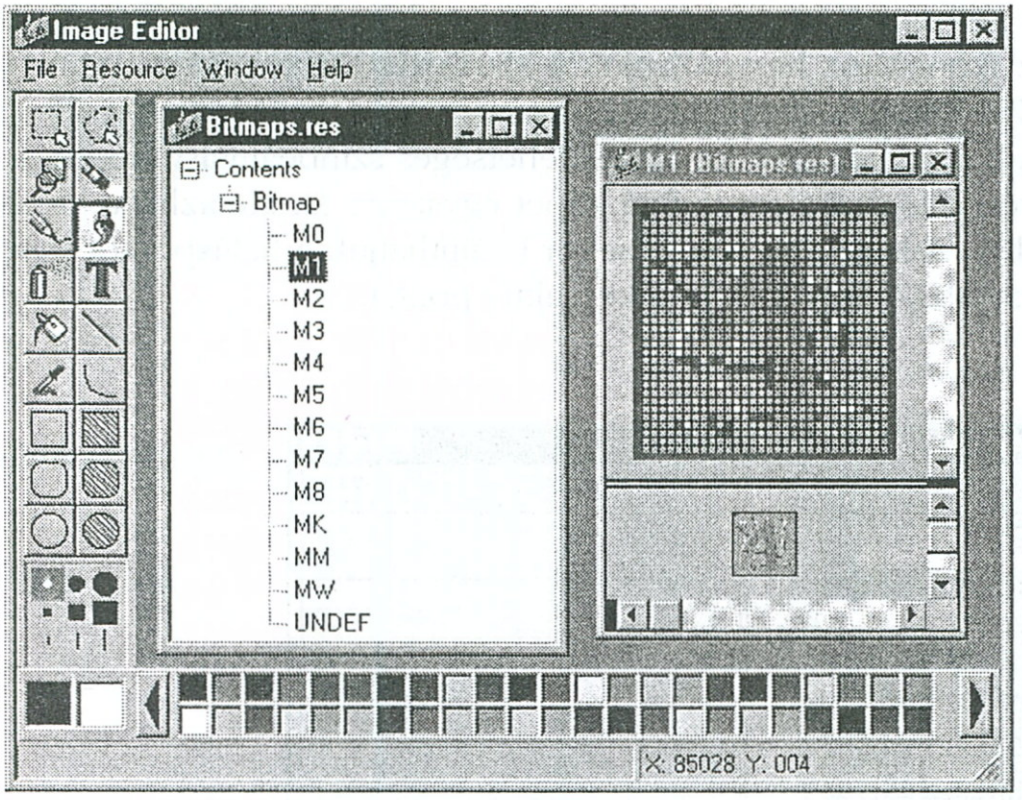
Az Image Editor felhasználásával bármilyen ikont, kurzort vagy bittérképet készíthetünk. Az ikonokhoz használhatjuk a szokásos 32x32 pixeles bitmap képet, vagy a Windows 95 16x16 pixeles formátumát (ezt láthatjuk a 27.1. ábrán). Ugyancsak használhatjuk az összes lehetséges színbeállítási módot is. Egyetlen ikon erőforrás az összes típusú képet egyszerre tartalmazhatja, nem csak egyet közülük. A kurzorok készítésekor beállíthatjuk a hatáspontot (hot spot) is, amely azt jelzi, hogy hol van a kép aktív pontja.



27.1. ábra *A Delphi Image Editor, egyetlen ikon erőforrás több, különböző típusú képével*

Az Image Editort alapvetően kétféle képen használhatjuk:

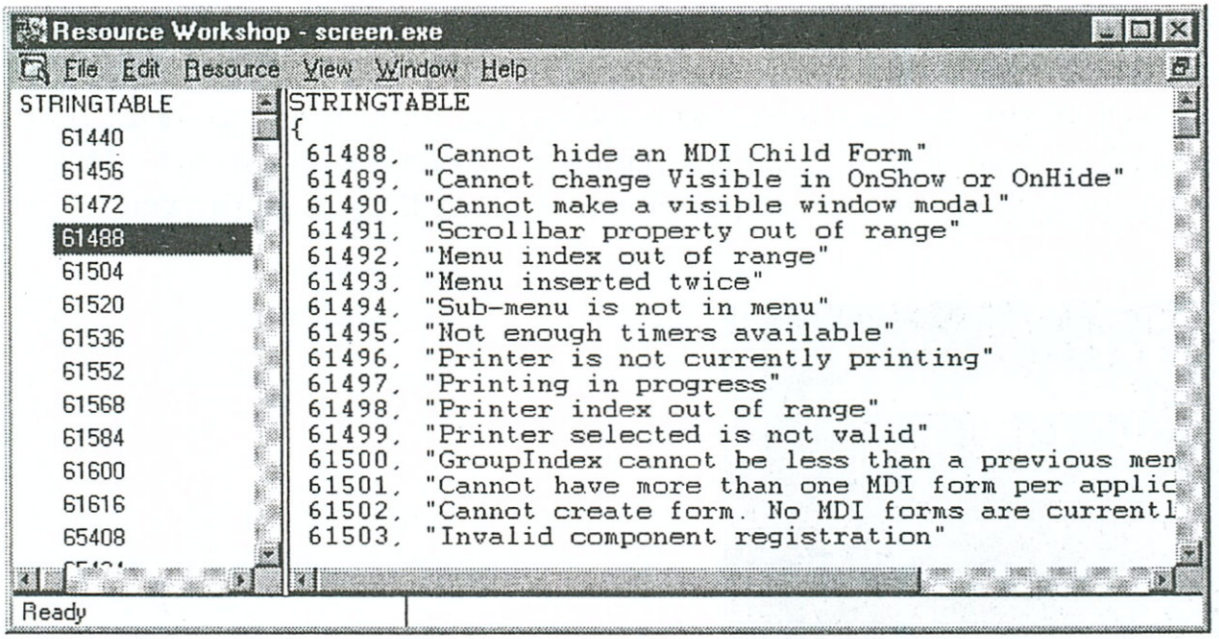
- Elkészíthetünk fájlokat azért, hogy tervezéskor (a megfelelő tulajdonságok beállításával) vagy futás alatt (a LoadFromFile metódusok segítségével) használjuk fel őket.
- Készíthetünk erőforrás fájlokat is, és ezeket futás közben Widows API hívásokkal érjük el. Ennek részleteit a következő fejezetben találhatjuk meg. Ha az Image Editorban készítjük grafikus erőforrásainkat, akkor jól áttekinthető formában láthatjuk az egyes csoportokba tartozó fájlok listáját.



27.2. ábra *Bittérkép ábrák listája egy erőforrás fájlban az Image Editor megjelenítésével (a 11. fejezet Mines példaprogramjából)*

Bár az Image Editor hasznos segítőtárs, nem képes az összes erőforrástípus kezelésére. Ha ilyen eszközre van szükségünk, akkor használhatjuk a Borland Resource Workshop programot (vagy más kereskedelmi forgalomban kapható erőforrás szerkesztőket). A Resource Workshop lehetőséget nyújt bármelyik típusú erőforrásfájl megnyitására és szerkesztésére. Ezzel a programmal már lefordított programok (DLL-ek, VBX-ek, stb.) erőforrásaihoz is hozzáférhetünk, de mielőtt ezeket az elemeket felhasználnánk, győződjünk meg róla, hogy a copyright jogok ezt lehetővé teszik számunkra.

Hogyha egy Delphiben készült alkalmazást nyitunk meg a Resource Workshoppal, tapasztalhatjuk, hogy valójában sok erőforrást tartalmaz a RES a fájlban található ikonon kívül. Alapértelmezésben egy Delphi futtatható fájl tartalmaz egy stringtáblát, melyben rendszerüzenetek, figyelmeztetések, és más általános stringek (pl. hónapnevek) találhatóak. Szintén találhatunk a fájlban saját erőforrásokat, amelyek bináris (Delphi által definiált RCDATA formátumban) a formokat írják le. Ezen kívül szerepel még néhány kurzor és egy ikon. A 27.3. ábra a 25. fejezet Screen példaprogramjának erőforráslistáját mutatja be, ami egy viszonylag egyszerű program. Bonyolultabb programokban jóval több erőforrás is szerepelhet, hogyha több formot, VCL egységet használunk, vagy további ikonokat, képeket is igénybe veszünk.



27.3. ábra *Egy lefordított Delphi alkalmazás erőforrás-listája a Resource Workshoppal (a 25. fejezet Screen példaprogramjából)*

Tipp

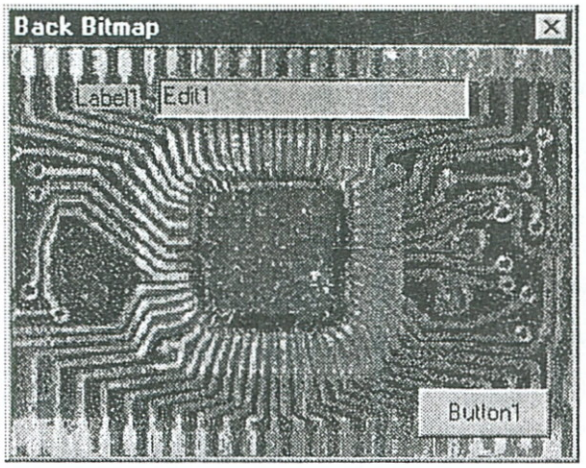
A Delphi példaprogramjai között található a Resource Explorer. Ezzel a programmal futtatható fájlokat nyithatunk meg (EXE vagy DLL), és megnézhetjük ezek legtöbb erőforrását, akárcsak a Resource Workshopban. A Resource Explorerben nem találunk beépített erőforrás szerkesztőket, de erőforrásokat a Vágólapra helyezhetünk segítségével, és innen szinte minden erőforrás szerkesztővel átemelhetjük saját Delphi alkalmazásunk erőforrás fájljába.

Ahogy már korábban említettem, Delphiben az erőforrásokat kétféleképpen használhatjuk fel: vagy egy komponens egyik tulajdonságában, vagy kifejezetten erőforrások formájában. Ebben az esetben néhány programsort mindenképpen írnunk kell.

Ikon vagy bittérkép felhasználása komponens tulajdonságaként

Az erőforrásokat legegyszerűbben a komponensek tulajdonságain keresztül tudjuk felhasználni. Egyelőre azonban csak ikon és bittérkép erőforrásokat kezelhetünk így, illetve bizonyos esetekben metafájlokat. Például, elhelyezhetünk egy Image komponenst egy párbeszédablak hátttereként, és ebben

megjeleníthetünk egy képet. A 27.4. ábra a BackBmp példaprogramot mutatja be, amelyben egy bittérképet használunk háttérként. Ebben az esetben a bittérkép az alkalmazás egy beágyazott erőforrása, ami azt jelenti, hogy nincs szükségünk az eredeti BMP fájlra (példánkban az eredeti bitmap fájl a Delphi Image könyvtárában van, ami alapértelmezésben az IMAGE alkönyvtár).



27.4. ábra *Egy bittérképes ábrát háttérként használó program, a BackBmp*

A BackBmp példában és hasonló programokban valójában a futtatható fájl nem tartalmazza a képet különálló bittérkép erőforrásként, hanem a form bináris leírásában szerepel. Ennek a módszernek az az előnye, hogy nem lehet olyan egyszerűen „kilopni" a bitképet a programunkból egy erőforrás szerkesztővel. Megjegyezem, hogy egy Delphi form is visszafejthető a Resource Workshoppal (vagy más, hozzá hasonló program segítségével). Ehhez először mentsük el az erőforrást RES formátumú fájlba (de DFM kiterjesztéssel), majd ezt a DFM fájlt nyissuk meg a Delphi szerkesztőjében. Ezzel a módszerrel egyszerűen csak azt kell tudnunk, hogy az alkalmazás Delphiben készült.

Ezzel szemben, hogyha a program ikonjait a szokásos erőforrás formátumban kapcsoljuk a lefordított fájlhoz, akkor bizonyos alkalmazások (például az Intéző vagy maga a Windows 95 shell) lehetővé teszik a felhasználó számára az erőforrások közvetlen elérését.

Az erőforrások közvetlen beolvasása

Az erőforrások elérésének második módja a Delphiben a közvetlen hozzáférés. Ebben az esetben először el kell készítenünk a különálló erőforrásfájlt a szükséges erőforrásokkal, majd ezt a fájlt be kell építenünk a projektbe. Ezt a $R fordítói direktívával tehetjük meg. A szokásos C/C++ megközelítéssel szemben, a Delphi projektek akár több erőforrásfájlt is tartalmazhatnak.

Figyelem!

Ne alakítsuk át az alapértelmezett erőforrás fájlt (amelyiknek ugyanaz a neve, mint a projektnek), mert a Delphi néha megváltoztatja ezt a fájlt, és ezzel elveszíthetjük módosításainkat. Nyugodtan készíthetünk viszont egy új RES fájlt a projekt könyvtárában, és egy fordítói direktívával betölthetjük azt.

Például a 11. fejezet Mines programjában (lásd a korábbi 27.2. ábrát) a bitképeket egy különálló erőforrásfájlból adtuk az alkalmazáshoz a form implementációs részében:

```
{$R *.DFM}
{$R BITMAPS.RES}
```

Ez a két sor kísértetiesen hasonlónak tűnik. Az első egy speciális erőforrásfájlt tölt be, aminek neve megegyezik a Pascal fájl nevével, de kiterjesztése DFM. Ez az a formfájl, amelyik megfelel az egységnek. Valójában, ebben a direktívában a csillag nem az összes DFM fájlt jelenti, hanem azt az egyet, amelynek neve megegyezik annak a fájlnak a nevével, amit éppen írunk. Ez is azt mutatja, hogy a Delphi lefordított form fájljait a rendszer egyéni erőforrásokként kezeli.

Hogyha mér elkészítettük az erőforrásokat, és beépítettük őket a programba, akkor a következő Windows API függvényeket használhatjuk kezelésükhöz: LoadAccelerators, LoadBitmap, LoadCursor, LoadIcon, LoadMenu, LoadResource és LoadString. A LoadResource kivételével minden egyes függvény egy bizonyos típusú erőforrás beolvasására alkalmas. A LoadResource egyéni erőforrásokhoz használható. Ezen eljárások első paramétere az alkalmazás-példány azonosítója, melyet Delphiben a HInstance globális változó tárol. A második paraméter a betöltendő erőforrás nevét tartalmazza. Természetesen, az alkalmazásoknak több ikonja, bitképe és egyéb erőforrása is lehet, ezért mindegyik elemre a nevével hivatkozhatunk. A LoadString függvénynek néhány további paramétere is van, amelyek meghatározzák, hogy hova töltsük be a stringet, illetve a beolvasandó szöveg hosszát.

A többi függvény egyszerűen egy azonosítót ad vissza, amely a beolvasott erőforrást határozza meg. Ezt az értéket közvetlenül felhasználhatjuk egy VCL objektum tulajdonságának megadására:

```
Bmp.Handle := LoadBitmap (HInstance, Name);
```

Ebben a sorban a Name egy nulla karakterrel lezárt string, egy PChar, ami a bitmap nevét tartalmazza, és a Bmp egy `TBitmap` osztálybeli objektum. Ez az utasítás a korábban bemutatott Mines példaprogram forráskódjából származik.

Tanulmányozzuk ezt a példát és leírását a 11. fejezetben, hogy megértsük a több bitmap erőforrás használatát és ennek előnyeit.

Mivel a korábbi példákban már használtunk bittérkép erőforrásokat (Például a World2 példaprogramban a 11. fejezetben, ahol számos képet betöltöttünk egy listába), most eltekintek további példák bemutatásától. Térjünk át inkább az ikonok, kurzorok, stringtáblák használatának ismertetésére.

Programok és formok ikonjai

A Delphiben minden egyes programnak és azok minden egyes formjának megvan a maga ikonja, amit az Icon tulajdonságban állíthatunk be. Ha egy form esetén nem állítjuk be ezt az értéket, akkor a program egyszerűen az alkalmazás objektum ikon tulajdonságát használja. Az alkalmazások alapértelmezett ikonját a Project Options párbeszédablak **Application** oldalán nézhetjük meg vagy írhatjuk át. Szintén ezt az ikont használja a Windows95 tálcája is, hiszen a Delphi alkalmazások tálcán megjelenő ablaka megegyezik a globális Application objektum ablakával.

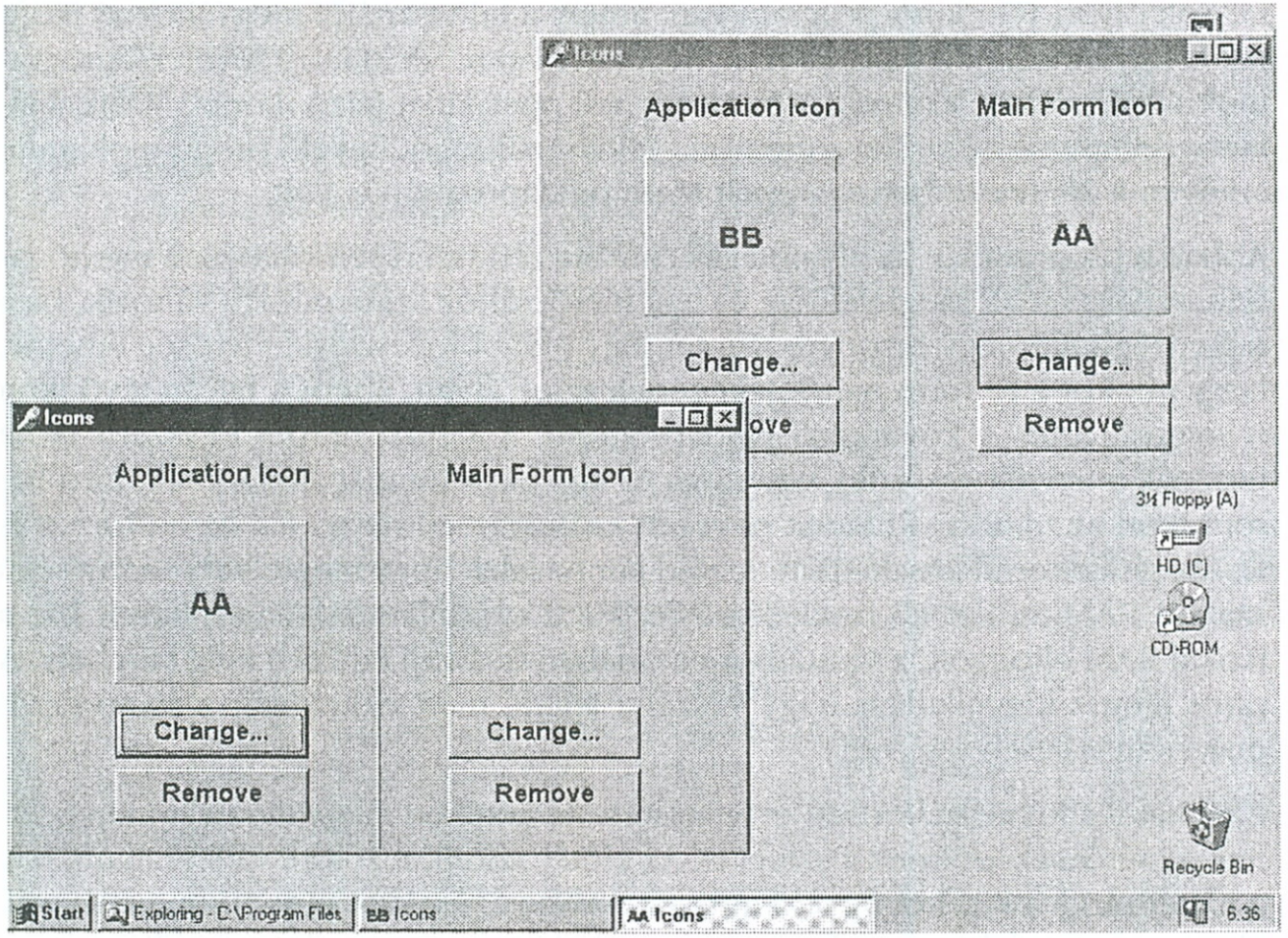
Ezeket a lehetőségeket az Icons példaprogramban demonstráljuk. A példa formja két részre osztható: baloldalon található egy címke, egy Image komponens és két nyomógomb, amelyek a form ikonjához kapcsolódnak. A jobb oldalon ugyanilyen komponensek vannak, de itt az alkalmazás ikonjához tartoznak. A két részt egy vonallal választottuk el. Szerepel a formon továbbá egy OpenDialog komponens, amit erőforrás fájlok keresésére fogunk felhasználni. Felhívnám a figyelmet arra, hogy külön definiáltam a form (a BB.ICO fájlban) és az alkalmazás (az AA.ICO fájlban) ikonját. A két ikon fájlt az Image Editor segítségével készítettem el. Valahányszor valamelyik Change nyomógombon kattintunk, egy új ikont olvasunk be egy külső fájlból:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with OpenDialog1 do
    if Execute then
    begin
      Application.Icon.LoadFromFile (Filename);
      Image1.Picture.LoadFromFile (Filename);
    end;
end;
```

Ha a Remove nyomógombok valamelyikét nyomjuk le, akkor a megfelelő ikont egyszerűen töröljük:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  Application.Icon := nil;
  Image1.Picture := nil;
end;
```

Ezzel a programmal tanulmányozhatjuk azt is, hogy a két Icon tulajdonság hogyan hat a lekicsinyítet alkalmazás-ikonra. Néhány példát mutat be a 27.5. ábra.



27.5. ábra *Az Icons példa futás közben*

Ha az Intézővel megnézzük az Icons alkalmazást, az Intéző automatikusan azt az ikont fogja használni, amit a form lekicsinyítésekor alkalmaztunk, nem pedig a fő form ikonját. Ez azért van így, mert a form ikonja a formot leíró egyéni erőforrásban (a DFM fájlban) el van rejtve, míg az alkalmazás-ikon az ikon erőforrásban kerül elhelyezésre, ami szabványos módon kötődik a futtatható fájlhoz.

Többféle ikon használata a megváltozott állapotok jelzésére

Az Icons programban az alkalmazás ikonjának megváltoztatásához külső fájlt használtunk. Ennek két hátránya is van:

- Az egyik, hogy az ikonfájlokra is szükség van a program futtatásához.

- A másik, hogy a külső fájlból történő olvasás egy kicsit tovább tart (bár ez erőforrások esetén nem sokat számít)

Hasznosnak találhatjuk a következő megoldást: adjuk az ikonokat a program erőforrásaihoz, és használjuk innen őket a többféle megjelenés megvalósítására. Sok olyan Windows program van, amely a felhasználót megváltozott ikonnal is figyelmezteti a megváltozott körülményekre. Például, a levelező programoknak általában van egy üres és egy teli postaládát jelző ikonja. Másik általános felhasználási terület a memória-felhasználtságot figyelő programok, mint amilyen a 26. fejezetben szerepelt Mem példaprogram is volt.

Abban a programban az alkalmazás címsora rendszeresen változik a memória-felhasználtságnak megfelelően, és egy mérőeszköz színe pirosra változik, ha a telítettség túllép egy bizonyos szintet. Egy továbbfejlesztési lehetőség lehetne, hogy amikor a fő form mérőeszközének színe átvált, akkor a tálcán lévő ikon is megváltozik. A következő példaprogram, a Mem2, három egyszerű ikont használ, amelyek egy zöld, egy sárga és egy piros lámpát ábrázolnak. Ez a három ikon az új, RYG.RES-nek nevezett erőforrás fájl része. Ezt az erőforrást is az Image Editorral készítettem. A zöld lámpa jelzi, hogy megfelelő mennyiségű szabad RAM-terület áll rendelkezésre. Ha a sárga lámpa világít, akkor bár a fizikai RAM elfogyott, a virtuális memóriában (a swap fájlban) még van hely. A piros lámpa kigyulladása nagy veszélyt jelent, mert ekkor már a swap fájl is majdnem teljesen tele van.

Ezenkívül a felhasználói felület fejlesztése érdekében megváltoztattam magukat a mérőeszközöknek a színét is. Az előtér színüket zöldre változtattam, míg a háttér sárga a RAM-ot figyelő műszer esetében, és piros virtuális memóriánál. Mindezt azért alakítottam át, mert az eredeti Mem programban, ha elfogyott az összes RAM, akkor a RAM figyelő eszköz majdnem teljesen fehér lett (csak a kerete volt piros színű). A módosított programban, ebben az esetben az ábra teljesen sárgává válik. Hasonlóan, ha a virtuális memória kezd elfogyni, akkor a Mem2 alkalmazás virtuális memóriát visszajelző mérőeszköze piros színűvé válik és ezzel hívja fel a figyelmet az aggasztó problémára.

A tálca alapértelmezett, zöld színű ikont a FormCreate metódus elején töltjük be futtatáskor:

```
procedure TMemForm.FormCreate(Sender: TObject);
begin
  Application.Icon.Handle:= LoadIcon(HInstance,'GREEN');
  {update the output ASAP}
  Timer1Timer (self);
end;
```

Figyeljük meg, hogy az alapértelmezett ikont azelőtt kell be olvasnunk, hogy a kezdeti értékeket kiszámolnánk az időzítő meghívásával, hiszen ellenkező esetben az OnTimer eseménykezelő nem tudná megjeleníteni a megfelelő

ikont, mert a betöltés egyszerűen mindig lecserélné a megfelelő ikont a zöldre. A Timer1Timer metódusból eltávolítottam a mérőeszközök színét megváltoztató kódot a korábban említett színcserék miatt, viszont belekerült az alkalmazás ablakának ikonját a memória-használtság alapján megfelelően beállító részlet. Ezek a sorok az eljárás végére kerültek:

```
procedure TMemForm.Timer1Timer(Sender: TObject);
var
  MemInfo : TMemoryStatus;
begin
  MemInfo.dwLength := Sizeof (MemInfo);
  GlobalMemoryStatus (MemInfo);
  RamGauge.Progress := MemInfo.dwAvailPhys div
    (MemInfo.dwTotalPhys div 100);
  VirtualGauge.Progress := MemInfo.dwAvailPageFile div
    (MemInfo.dwTotalPageFile div 100);
  Caption := Format ('Mem: = %d - %d',
    [RamGauge.Progress, VirtualGauge.Progress]);
  {copy the form caption to the application title on
   the taskbar}
  Application.Title := Caption;
  {set icon color}
  if RamGauge.Progress > 5 then
   Application.Icon.Handle:=LoadIcon(HInstance,'GREEN')
  else if VirtualGauge.Progress > 20 then
   Application.Icon.Handle:=LoadIcon(HInstance,'YELLOW')
  else
   Application.Icon.Handle:=LoadIcon(HInstance,'RED');
end;
```

Ez a program tökéletesen működik, de, ahogy azt már a 26. fejezetben, az eredeti program készítésénél is leírtam, sokkal szebb megoldás lenne a változó ikont a tálca ikontartójában megjeleníteni. Most ezt fogjuk megvalósítani.

A Tálca ikontartójának használata

A Windows 95-ben a Tálcával együtt, egy újabb területet is bevezettek a rendszerinformációk megjelenítéséhez, az ikontárolót. Ahogy a DOS-os időkben voltak TSR programok és a Windows 3.1 alatt néhány program csak ikon állapotban futott, Windows 95-ben bizonyos programok az ikontartóban futnak. A képernyő jobb alsó sarkában, az óra mellett található egy terület (a tálca ikontartója), ahol segédprogramjaink jelenhetnek meg.

Ennek használatához egyetlen API hívás áll rendelkezésünkre, a Shell_NotifyIcon. Ez egy nagyon egyszerű eljárás. Két paramétere van: az első egy pointer, amely egy TNotifyIconData struktúrára mutat, a második azt

határozza meg, hogy a tartóba berakni, onnan eltávolítani vagy megváltoztatni akarjuk az ikont. A rekord felépítése a következő:

```
type
  TNotifyIconData = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..63] of AnsiChar;
  end;
```

A TNotifyIconData rekord mezőinek jelentése a következő:

- `cbSize` a rekord méretét határozza meg (a rendszer ez alapján a verziószámot határozza meg).
- `hWnd` annak az ablaknak az azonosítója, amelyet értesíteni kell az ikon eseményeiről.
- `uID` az ikon azonosítója és akkor használható, ha alkalmazásunk több ikont is elhelyez a tartóban.
- `uFlags` három értéket vehet fel: `nif_Message`, `nif_Icon` és `nif_Tip`. Azt határozzák meg, hogy a struktúra utolsó három mezője közül, melyiket kell használni.
- `uCallbackMessage` annak a felhasználó által definiált üzenetnek az azonosítószámát tartalmazza, amelyen keresztül az ablakot (hWnd) értesíteni kell az ikonon történt felhasználói eseményekről.
- `hIcon` az ikontartóban megjelenítendő ikon.
- `szTip` a tipp szövege, amelyet akkor kell megjeleníteni, amikor az egér megáll az ikon felett.

Gyakorlatilag a rekord mezői (az ikon és a tipp kivételével) az értesítendő ablakot és az üzenetet határozzák meg. Tehát valójában amikor a felhasználó a tartóban lévő ikont igénybe veszi, az üzenetet küld a programnak az alkalmazás által definiált függvénnyel, és az eseménynek megfelelő paraméterekkel. Az alkalmazás megkapja az ikon azonosítóját is.

A memóriaállapot megjelenítése az ikontartóban

Az ikontartóról előbb megismert információk alapján most a Mem2 példaprogramot fejlesztjük tovább. Az új programot nevezzük Mem3-nak. A több, kisebb változtatás közül elsőnek említeném, hogy a két címkemezőben rögzített szövegek helyett, a szabad és az összes memória mennyiségét jelezzük vissza,

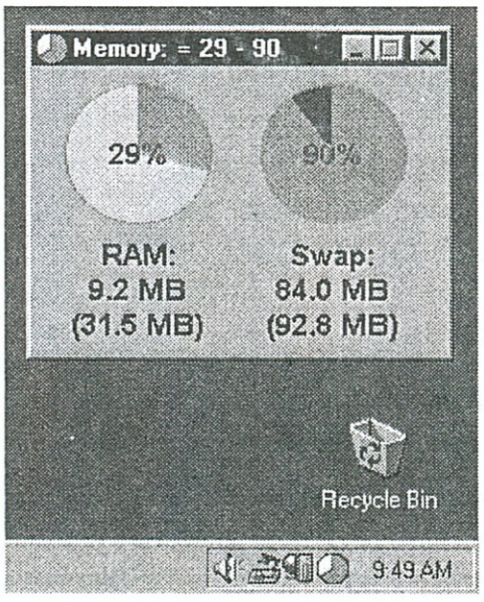
hogy még több információt adjunk a felhasználónak. Az első címke frissítésére használt kód, ezek alapján, a következő lett:

```
Label1.Caption := Format ('RAM:'+#13+'%s'+#13+'(%s)',
  [FmtMem (MemInfo.dwAvailPhys),
  FmtMem (MemInfo.dwTotalPhys)]);
```

Az FmtMem függvény egy saját rutin, amelyet az információk könnyebb elérése érdekében készítettem. Az eljárás a visszaadott szöveg végére elhelyezi a „KB" vagy a „MB" stringet, az érték nagyságának megfelelően:

```
function FmtMem (N: Integer): string;
begin
  if N > 1024*1024 then
    FmtMem := Format ('%.1f MB', [n / (1024*1024)])
  else
    FmtMem := Format ('%.1f KB', [n / 1024]);
end;
```

A 27.6. ábrán a megváltozott megjelenésről láthatunk példát. Mégis a legfontosabb változtatás a programon, hogy a program, indulásakor egy ikont helyez el az ikontartóban, és az időzítő-intervallumok lejártakor mindig frissíti is azt.



27.6. ábra *A Mem3 példaprogram kimenete: a két címkében több adat látható a memória állapotáról, és egy ikon található az ikontartóban*

Először is az egység `interface` részének `uses` utasításában a ShellApi egységet is felvettem. Ezután a form osztály mezői közé felvettem egy `TNotifyIconData` rekordot:

```
private
    nid: TNotifyIconData;
```

A form készítésekor a program inicializálja az adatstruktúrát, majd meghívja a Shell_NotifyIcon API függvényt, és ezzel elhelyezi az ikont az ikontartóban:

```
procedure TMemForm.FormCreate(Sender: TObject);
begin
  // load the initial icon
  Icon.Handle :=
    LoadIcon (HInstance, 'GREEN');
  // fill the NotifyIcon data structure
  nid.cbSize := sizeof (nid);
  nid.wnd := Handle;
  nid.uID := 1; // icon ID
  nid.uCallBAckMessage := wm_IconMessage;
  nid.hIcon := Icon.Handle;
  nid.szTip := 'Free memory';
  nid.uFlags := nif_Message or
    nif_Icon or nif_Tip;
  Shell_NotifyIcon (NIM_ADD, @nid);
  // update the output ASAP
  Timer1Timer (self);
end;
```

Az OnTimer eseménykezelőben lefut a korábbi verzió eljárása és frissíti az ikontartóban lévő ábrát a következő sorokkal:

```
// update the NotifyIcon structure
  nid.hIcon := Icon.Handle;
  strcopy (nid.szTip, PChar(Caption));
  nid.uFlags := nif_Icon or nif_Tip;
  Shell_NotifyIcon (NIM_MODIFY, @nid);
```

Ez a kódrészlet egyszerűen frissíti a `nid` rekord ikonját és feliratát (egy PChar típuskonverzió felhasználásával), majd ténylegesen megváltoztatja az ikont a tartóban. Ebben a metódusban a rekordnak csak azokat a mezőit frissítjük, amelyek megváltoztak (ezért vettük fel a struktúrát a form lokális mezői között). Amikor a program véget ér, az OnDestroy eseménykezelőben a Shell_NotifyIcon újabb meghívásával eltávolítjuk az ikont:

```
procedure TMemForm.FormDestroy(Sender: TObject);
begin
  nid.uFlags := 0;
  Shell_NotifyIcon (NIM_DELETE, @nid);
end;
```

Hogyha elfelejtjük eltávolítani az ikont, akkor az akkor fog eltűnni, amikor az egér először kerül fölé, de természetesen sokkal jobb megoldás az alkalmazás lezárásakor törölni.

A fő ablak elrejtése és megjelenítése

Az előbbi program fő ablakának csak akkor kéne megjelennie, amikor a felhasználó duplán kattint a tartóban lévő ikonon. Ezt nem túlságosan nehéz megvalósítani. Mindössze egyetlen sort kell a projektbe írnunk:

```
Application.ShowMainForm := False;
```

mielőtt a CreateForm metódust meghívjuk. Amikor a felhasználó bármelyik egérgombbal kattint a tartóban lévő ikonon, a form egy wm_IconMessage üzenetet kap. Ezt a formban a következő eljárással kezeljük le:

```
public
    procedure IconTray (var Msg: TMessage);
      message wm_IconMessage;
```

Ez a metódus egy előugró menüt jelenít meg, ha a jobb gombot nyomták le, vagy egyszerűen meghívja az alapértelmezett menükezelőt (Details1Click):

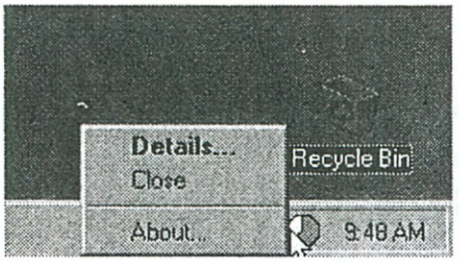
```
procedure TMemForm.IconTray (var Msg: TMessage);
var
  Pt: TPoint;
begin
  if Msg.lParam = wm_rbuttondown then
  begin
    GetCursorPos (Pt);
    SetForegroundWindow (Handle);
    PopupMenu1.Popup (Pt.x, Pt.y);
  end;
  if Msg.lParam = wm_lbuttondblclk then
    Details1Click (self);
end;
```

A SetForegroundWindow meghívásával érjük el, hogy az előugró menü megfelelően jelenjen meg (tehát minden más felett). Érdemes kipróbálni, hogy mi történik, ha ezt a sort kihagyjuk a programból: az ikon előugró menüje akkor is a képernyőn marad, amikor egy másik programra váltunk át. Ez a két választási lehetőség (a jobb kattintás a helyi menühöz és a bal dupla kattintás az alapértelmezett tevékenységhez) kötelező erejű vezérelv a Windows felhasználói felületen.

```
procedure TMemForm.Details1Click(Sender: TObject);
begin
  ShowWindow (Handle, sw_ShowNormal);
  SetForegroundWindow (Handle);
  Show;
end;
```

A Details1Click metódus a form egy előugró-menü komponensének alapértelmezett tevékenységét (Details) hívja meg. Ezen kívül a menüben szerepel még

a Close és az About pont. Az előugró menü nem a formhoz van kapcsolva, hanem akkor jelenik meg, amikor a felhasználó, a jobb gombbal kattint az ikonon. Ezt a 27.7. ábrán láthatjuk.



27.7. ábra *A Mem3 példaprogram ikonja és előugró menüje a tartóban*

A SetForegroundWindow függvényt tehát meg kell hívnunk, amikor egy másik alkalmazás aktív és a felhasználó kattint az ikonon. Ugyancsak szükség van a Delphi Show metódusának meghívására, melynek következményeként a Delphi létrehozza a form ablakát és a benne elhelyezendő elemeket. Ebben a példában, ha nem hívjuk meg ezt a függvényt, akkor is minden tökéletesen fog működni, de ha egy nyomógombot is elhelyeztünk volna a formon, akkor az egyáltalán nem jelent volna meg e nélkül a hívás nélkül.

A Close menüpont lezárja az alkalmazást:

```
procedure TMemForm.Close1Click(Sender: TObject);
begin
  Application.Terminate;
end;
```

A form lezárása helyett a Terminate metódust hívjuk meg, mert amikor a felhasználó a formot zárja le (a rendszermenüből, vagy a jobb felső sarokban lévő gombbal), akkor csak elrejteni akarjuk a formot. Ezért készült úgy a fenti lezáró rutin, és ezért néz ki így a FormClose eseménykezelő:

```
procedure TMemForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caNone;
  ShowWindow (Handle, sw_Hide);
end;
```

Különös, hogy ha az Action változót caHide értékre állítjuk, az leállítja a programot. Ezért ShowWindow függvénnyel vagyunk kénytelenek elrejteni a formot. Felhívom a figyelmet arra, hogy amikor meghívjuk az Application Terminate metódusát, akkor a form OnClose eseménykezelője nem fut le, az OnDestroy viszont igen.

A tartóban lévő ikon elrejtése

Most már el tudunk helyezni ikont az ikontartóban, egy időzítő segítségével meg tudjuk változtatni azt, hogyha a rendszer állapota megváltozik, és lehetővé tudjuk tenni azt is, hogy a felhasználó az ikonhoz kapcsolt előugró menün keresztül parancsokat adjon. Az előugró menü megjelenítése előtt meghívva a SetForegroundWindow függvényt ki tudtuk javítani a Windows 95 működési hibáját. Egyetlen nagy probléma van még hátra: amikor a program fut, a fő form rejtve van, de az alkalmazás-ablak jelenléte még látható a tálcán is. Az ilyen jellegű alkalmazásokhoz ez nem illik.

Hogyan tudnánk eltüntetni az alkalmazást reprezentáló ikont a tálcáról? Megtehetnénk, hogy egyszerűen elrejtjük a hozzá tartozó ablakot (Application.Handle), de ez azt jelenti, hogy először elkészítjük és megjelenítjük az ikont, és utána törölnénk azt. Valószínűleg csak egy rövid villanást látnánk a képernyőn, de mégsem tökéletes ez a megoldás.

Szebb megoldás, hogyha az alkalmazás-ablak felépítését akadályozzuk meg. Ezt úgy tehetjük meg, hogy a globális IsLibrary változót igazra állítjuk. Ha megnézzük a `TApplication` Create nevű konstruktorát, a következőt láthatjuk:

```
if not IsLibrary then
  CreateHandle;
```

Tehát a Delphivel azt hitetjük, hogy programunk egy DLL és így nem igényel fő ablakot. Ez egy elég veszélyes trükk, és a program lezárásakor igen érdekes viselkedést eredményez, miszerint nem hívódik meg az OnDestroy eseménykezelő. Ezért úgy döntöttem, hogy bár az IsLibrary változót igazra állítjuk, de amint lehet visszaírjuk a hamis értéket. Az igazi problémát most az jelenti, hogy a mezőt az előtt kell igazra állítani, mielőtt a globális Application objektum létrejönne. Ez a Controls egység inicializáló részében történik meg. Tehát most az a kérdés, hogy hogyan tudnánk egy egység inicializálása előtt saját kódot futtatni. Az alábbi sorok beírása nem segít rajtunk:

```
IsLibrary := True;
Application.CreateForm(TMemForm, MemForm);
Application.Run;
```

A Delphi a egységek inicializálását azelőtt elvégzi, hogy a projekt kódjára ránézne. A megoldást az jelenti, hogy a Delphi olyan sorrendben inicializálja a egységeket, amilyen sorrendben a `uses` részben soroljuk fel őket. Alapértelmezésben ez a sorrend a következő:

```
uses
  Forms,
  Resform in 'RESFORM.PAS' {MemForm};
```

A ResForm egység valójában már felhasználja a Forms egységet (ami viszont behívja a Controls egységet is), így már a ResForm inicializáló részében is késő

elvégezni a szükséges változtatásokat. A probléma megoldásához egy új egységet írtam, a következő, közérthető tartalommal:

```
unit RunFirst;
interface
implementation
initialization
  IsLibrary := True;
end.
```

Ezek után a projekt forráskódját az alábbira módosítottam. Figyeljük meg, hogy a RunFirst egységet soroljuk fel először (így ez inicializálódik elsőként), és, hogy az IsLibrary változót azonnal visszaállítjuk hamis értékűre. Végeredményben tehát, így néz ki a projekt forráskódja:

```
program Mem;
uses
  RunFirst in 'RunFirst.pas',
  Forms, Windows,
  Resform in 'RESFORM.PAS' {MemForm};
{$R *.RES}
begin
  Application.ShowMainForm := False;
  IsLibrary := False;
  Application.CreateForm(TMemForm, MemForm);
  Application.Run;
end.
```

Nem esküdnék meg rá, de én még egyetlen hátrányát sem tapasztaltam (főleg egy ilyen kis program esetén) annak, hogy megakadályozzuk az alkalmazásablak elkészítését. Az így elkészült programban, amint először megjelenik a fő form, ikonja is azonnal a tálcára kerül.

Olyan alkalmazások készítése, melyek használják az ikontartót, valószínűleg a legegyszerűbbek közé tartozik azok közül, melyek a Windows95 (vagy a Windows NT 4.0) keretprogram kezelőfelületét alakítják át. Sok más lehetőséggel is próbálkozhatunk, például a 21. fejezet programja is kiterjeszti a keretprogramot. A Delphi majdnem mindent lehetővé tesz, de néha olyan akadályokba ütközünk, melyeket kénytelenek vagyunk átugrani vagy megkerülni a megfelelő eredmény elérése érdekében. Talán szükségtelen mondanom, hogy ezen akadályok leküzdése sok örömöt jelent, s hozzájárulnak a Delphi filozófiájának alapos megismeréséhez.

A kurzor használata a Delphiben

A Delphi igen széles körben támogatja a kurzorok használatát. Így jóval kevesebb munkát igényelnek, mint az ikonok kezelése. Például a Delphiben sok előre definiált kurzor létezik. Ezek közül némelyik Windows alapértelmezés,

de a legtöbbjük a Delphi saját készletéhez tartozik. Delphiben a kurzorok használata egyértelmű: egyszerűen használjuk az Object Inspectort egy komponens Cursor vagy DragCursor tulajdonságának megfelelő értékre állításához. Ha egy bizonyos ideig az egész alkalmazásban ugyanazt a globális kurzort kívánjuk használni, akkor a Screen komponens Cursor tulajdonságát kell beállítanunk.

Bár itt nincs lehetőségünk az Object Inspector használatára, néhány programsorral célt érhetünk. A következő kódrészlet a homokóra kurzor beállítására általánosan használt módszert mutatja be, amit akkor használunk, ha programunk olyan tevékenységbe kezdett, ami sokáig nem igényel felhasználói beavatkozást:

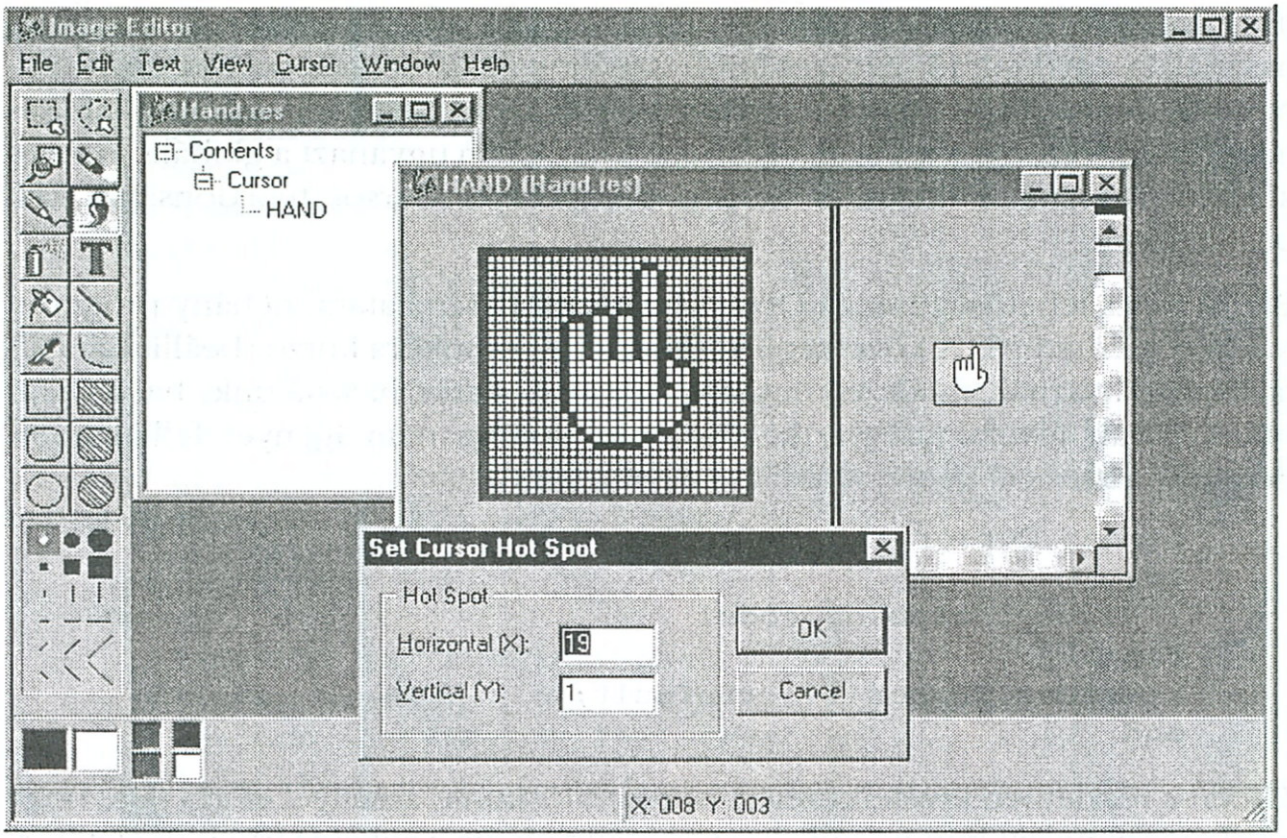
```
Screen.Cursor := crHourglass;
try
  {time-consuming code}
finally
  Screen.Cursor := crDefault;
end;
```

Ebben a részletben kivételkezelést is megvalósítunk, amellyel elérhetjük, hogy az alapértelmezett kurzort akkor is visszaállítjuk, ha a sokáig tartó művelet közben valamilyen hiba történik.

A form, a komponensek és a Screen objektum Cursor tulajdonsága is `TCursor` típusú. Meglepetéssel szolgálhat, ha a Delphi súgójában megnézzük a `TCursor` típus definícióját, ugyanis ez nem egy osztály, hanem mindössze egyetlen numerikus érték. A `TCursor` egy integer érték, amelyet a Delphi egy olyan tömb indexeként használ, melyben kurzor-kezelők vannak felsorolva. Ez a tömb a Screen objektum Cursors (s-sel a végén) tulajdonságán keresztül érhető el. Ugyancsak ez a tömb ad lehetőséget felhasználói kurzorok betöltésére az alkalmazás erőforrásai közül.

Saját kurzor készítése

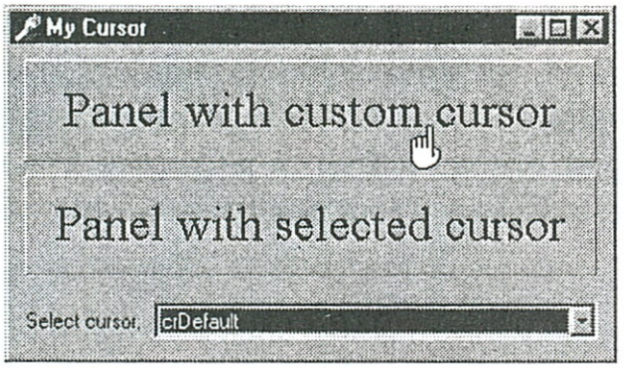
A következő példa, melyet MyCur-nak neveztem el, azt mutatja be, hogyan használhatjuk programjainkban saját kurzorjainkat, és, hogy hogyan kezeljük a Screen objektum Cursors tulajdonságát általában. A példa egy új erőforrás fájlt használ, amelyben definiálja a 27.8. ábrán látható kéz kurzort. Ugyancsak ezen az ábrán látható a kurzor hatáspont párbeszéd-ablaka. Ha új kurzort készítünk, ügyeljünk rá, hogy el ne felejtsük beállítani annak hatáspontját. Ez egyetlen pixel a kurzor határain belül, és ez jelenti az egér pontos pozícióját. Hogyha nem itt akarjuk kiélni művészi hajlamainkat, érdemes végignézni a Delphi Image Library (alapértelmezésben az IMAGES\CURSORS mappát jelenti) könyvtárában található kurzorokat.



27.8. ábra *Ha saját kurzor erőforrást készítünk, ne felejtsük el beállítani a hatáspontját!*

A HAND.RES nevezetű erőforrás fájlt a {$R HAND.RES} direktívával beépítjük programunkba. A példa formján két panel és egy kombinált lista található. Egy listában megjeleníti az elérhető kurzorokat. A 27.9. ábrán láthatjuk a teljes formot, itt pedig a kombinált lista szöveges leírását láthatjuk:

```
object ComboBox1: TComboBox
    Style = csDropDownList
    Items.Strings = (
      'crHand (custom cursor)'
      'crDefault'
      'crNone'
      'crArrow'
      'crCross'
      ...)
    OnChange = ComboBox1Change
  end
```



27.9. ábra *A MyCur példaprogram formja: két panel és egy kombinált listában a lehetséges kurzorok*

A program kezdetén, a FormCreate metódus tölti be a felhasználói kurzort az alkalmazás erőforrásai közül, a LoadCursor API függvényhívással. Ezek után ezt a kurzort a Cursors tömbben tároljuk, a felhasználó által definiált crHand konstans segítségével. Ennek értékét 1-re állítottuk. Pozitív értékek használatával elkerülhetjük az előre definiált kurzorok felülírását, mert ezek csak negatív számokat és a 0-t használják.(ezt hamarosan részletesebben megnézzük) Hogyha sikerült a tömbbe másolnunk a kurzort, akkor mostantól kezdve ugyanúgy használhatjuk mint az összes többi kurzort. Ennek megvalósítása, ezek alapján, a következő:

```
{$R HAND.RES}
const
  crHand = 1;
procedure TForm1.FormCreate(Sender: TObject);
begin
  {load a custom cursor from the resources}
  Screen.Cursors [crHand] := LoadCursor (
    HInstance, 'HAND');
  {use the new cursor for the first panel}
  Panel1.Cursor := crHand;
  {select the crDefault line}
  ComboBox1.ItemIndex := 1;
end;
```

A fenti rutinban összesen két sor végzi az érdekes feladatot. A második panel kurzorának kiválasztása a kombinált lista segítségével még egyszerűbb, hiszen egyetlen sorral megoldható:

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  Panel2.Cursor := 1 - ComboBox1.ItemIndex;
end;
```

Ugye milyen furcsa? Ahogy már korábban is említettem, a Screen objektum Cursors tulajdonsága egy olyan tömböt tárol, melynek minden tagja egy-egy kurzor kezelőt tárol. A TCursor típus, amellyel a Cursor tulajdonságokat megadjuk, valójában integer típus. Ezt az egész értéket a Delphi a Cursors tömb indexeként használja fel a kívánt kurzor eléréséhez. Tehát amikor a következő sort írjuk:

```
Form1.Cursor := crArrow;
```

akkor a Delphi ezt így alakítja át:

```
Form1.Cursor := -3;
```

Pontosan ezt tettük a felhasználói kurzor definiálásakor is, hiszen ott is egy konstanst definiáltunk egy kurzor elérésére a `Screen.Cursors` tömbben. Így már érthetővé válik a fenti bonyolult utasítás. Az összes kurzor szöveges azonosítóját felvettem a kombinált lista Items tulajdonságában, mégpedig ugyanabban a sorrendben, mint a megfelelő konstans értékek, pontosabban éppen fordított sorrendben, mert a Delphi alapértelmezett kurzorainak negatív értékek felelnek meg, 0-tól -20-ig. Mivel a saját kurzoromat adtam a listához elsőként, így a lista első eleme, melynek indexe (ItemIndex) 0, az 1-es kurzort jelöli ki, a második elem (ItemIndex=1 értékkel) a 0-s kurzorra, a harmadik (ItemIndex=2-vel) a -1-es kurzorra hivatkozik, és így tovább.

Változtatható kurzor

Az előző részben tárgyalt MyCur példaprogramban kiválaszthatjuk a crNone kurzort, pedig ez nem szerepel az Object Inspector listájában, mint megfelelő kurzor érték. Ez a hiány azzal magyarázható, hogy ezt, vagy bármely a Screen.Cursors tömb határain kívül eső értéket kiválasztva, a kurzor eltűnik. Hogyha egy komponens Cursor tulajdonságában ilyen értéket állítunk be, akkor amikor az egér e fölé a komponens fölé kerül, a kurzor eltűnik, és csak akkor jelenik meg újra, ha lekerül az elemről.

Ez a viselkedés azért lehetséges, mert a Windows lehetővé teszi, hogy egy ablak bizonyos területein más-más kurzort állítsunk be. Valójában, ha egy formnak vagy egy komponensnek nincs beállítva semmilyen kurzor sem, akkor beállíthatjuk azt saját kezűleg egy `wm_SetCursor` eseménykezelővel, ahogy azt a szokásos Windows programokban tehettük.

A következő példa, nevezetesen a FlexCur, a fent leírtakat mutatja be. Ennek az alkalmazásnak a formját egyetlen, üres Image komponens tölti ki. A komponens Canvas tulajdonsága a memóriában tárolt bittérképre mutat, így az ablak felülete automatikusan újrafesthető. Az Image komponens majdnem teljesen kitölti a felhasználói területet, mégsem teljesen fedi el azt, mert később megmagyarázott okokból, egy vékony üres keretet szabadon hagyunk.

Egy változó kurzor használatának bemutatásához egy kör-rajzoló programot írunk, amellyel törölhetünk is majd. Ha futás közben a formra kattintunk, akkor ezzel egy kis kört rajzolunk rá, illetve ha egy kör belsejében bármelyik pixelen kattintunk, a kör eltűnik. A kattintás által kiváltott tevékenységet kétféle kurzorral jelezzük. Tehát a kurzor állapota megváltozik, ha egy már korábban kirajzolt kör fölé kerül, jelezvén, hogy kattintáskor törölni fogunk.

Ennek végrehajtásához, a program a form kurzorját crNone értékre állítja, és lekezeli a `wm_SetCursor` üzenetet:

```
public
    procedure WmSetCursor (var Msg: TWMSetCursor);
      message wm_SetCursor;
```

Ebben az üzenet-kezelőben két dolgot vizsgálunk meg. Először, ellenőriznünk kell, hogy az egér a felhasználói területen van-e, vagy a kereten. Ezt az üzenet HitTest paraméterének segítségével és a htClient érték ellenőrzésével állapíthatjuk meg, mely pontosan azt jelzi, hogy a kurzor a felhasználói területen van-e (a HitTest kódok listáját a Windows API súgófájlban a `wm_NCHitTest` üzenetnél találhatjuk meg).

Megjegyzés

A Windows mindannyiszor elküldi a `wm_SetCursor` üzenetet az ablaknak, valahányszor az egér pozíciója megváltozik az ablakban. A FlexCur program ez az üzenetet figyeli. Alapértelmezésben a Delphi a formok és komponensek Cursor tulajdonságát használja arra, hogy megállapítsa, hogy az adott pozíción milyen legyen a kurzor. Ez a viselkedés könnyen megváltoztatható. Például egy egyszerű megoldás, hogy az OnMouseMove kezelőt használjuk a kurzor beállítására. Ekkor azonban a kurzor villogni fog, mert a rendszer, közvetlenül azelőtt, hogy mi beállítjuk, az alapértelmezett kurzort (crNone) jeleníti meg.

Ha nem a felhasználói területen állunk, akkor a szülő üzenetkezelőjét futtatjuk az `inherited` kulcsszó segítségével. Ezzel az alapértelmezett Delphi formok viselkedését hajtjuk végre. Viszont, hogyha a kurzor a felhasználói területen van, akkor megvizsgáljuk az egér alatti pont színét. Mivel a `wm_SetCursor` üzenetben nem kapjuk meg paraméterként a kurzor koordinátáit, a GetCurPos API hívással állapítjuk meg ezt. A függvény a képernyő koordinátarendszere szerinti értékeket ad vissza, így azokat átkonvertáljuk az Image komponens koordinátáivá, és ezeket tudjuk felhasználni az adott pont színének

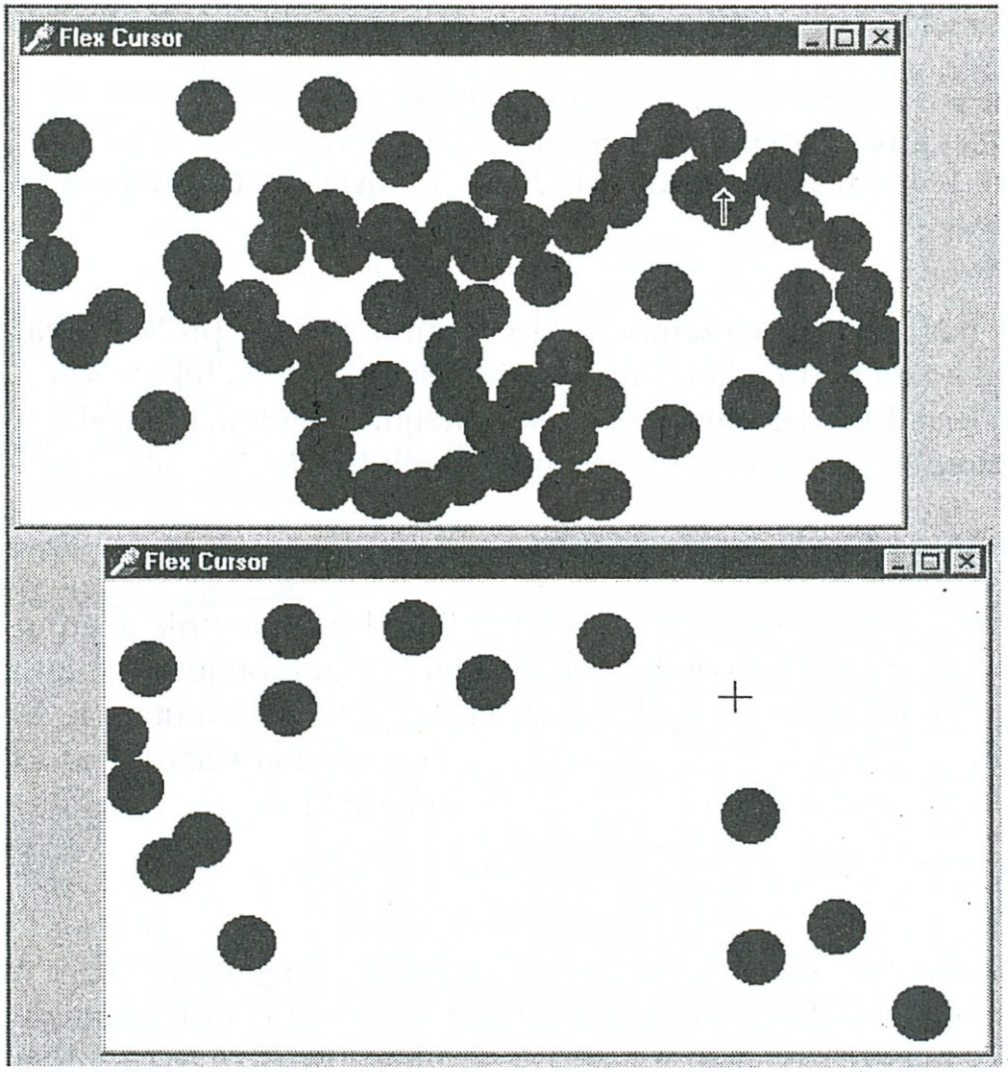
megállapítására. A színt a komponens Canvas tulajdonságának Pixels tömbjéből olvashatjuk ki. Ezek után lássuk egyben az eseménykezelő eljárás listáját:

```
procedure TForm1.WmSetCursor (var Msg: TWMSetCursor);
var
  CurPos: TPoint;
begin
  {get the position of the cursor, and
  convert it to client coordinates}
  GetCursorPos (CurPos);
  CurPos := Image1.ScreenToClient (CurPos);
  if Msg.HitTest = htClient then
    if Image1.Canvas.Pixels [CurPos.X,
        CurPos.Y] = clWhite then
      {if the pixel is white}
      SetCursor (Screen.Cursors [crCross])
    else
      {if the pixel is black}
      SetCursor (Screen.Cursors [crUpArrow])
  else
    {if the mouse is outside the client area}
    inherited;
  Msg.Result := 1;
end;
```

Ahogy látható, a színvizsgálat eredményétől függően, a program meghívja a Windows API SetCursor függvényét, de paraméterként nem a Delphi által definiált konstanst (crCross vagy crUpArrow), hanem magát a kurzort adjuk át. A konstansokból úgy tudjuk a valódi kurzorokat elérni, hogy kiolvassuk azokat a Screen objektum Cursors tulajdonságából, a tömb megfelelő elemet kiválasztva.

Figyeljük meg, hogy a fenti függvény utolsó sora az üzenetkezelő által visszaadandó értéket állítja be (`Msg.Result:=1`). Bár ritkán használjuk Windows üzenetek visszatérési értékét, ha kihagyjuk, amikor szükség van rá, akkor előfordulhat, hogy a rendszer az általunk már lekezelt üzenetet újra megpróbálja feldolgozni. Ezzel az üzenettel például, azt kell jeleznünk a Windowsnak, hogy már nem kell az alapértelmezett kurzort beállítani, mert mi már elvégeztük a beállítást.

A program a kereszt (crCross) kurzort használja a fehér területek jelzésére, és a felfelé mutató nyilat (crUpArrow) a fekete területekhez. A nyilat itt a festés művelet képi megjelenítésére használjuk, amellyel a színezett területet törölhetjük. Ezek a törlések nem csak egyetlen kört törölnek, hanem az összes ehhez érintkezőt is. A 27.10. ábrán ennek megfelelően két egymás utáni képet láthatunk.



27.10. ábra *A FloodFill metódus egyszerre több kört töröl a FlexCur példaprogramban. Ez látható, ha a két képet összehasonlítjuk, hiszen az alsó képet egyetlen kattintással kapjuk. Figyeljük meg a kétféle kurzort!*

Ha a kattintás pillanatában az egér egy már fekete pixelen áll, akkor az Image1MouseDown eseménykezelőben az alapértelmezett Ellipse metódus helyett a FloodFill-t hívjuk meg:

```
procedure TForm1.Image1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  if Image1.Canvas.Pixels [X, Y] = clWhite then
  begin
    {paint a black circle}
    Image1.Canvas.Pen.Color := clBlack;
    Image1.Canvas.Brush.Color := clBlack;
    Image1.Canvas.Ellipse (X-15, Y-15, X+15, Y+15);
  end
```

```
    else
    begin
      {erases the contiguous black pixels}
      Image1.Canvas.Brush.Color := clWhite;
      Image1.Canvas.FloodFill (X, Y, clWhite, fsBorder);
    end;
  end;
```

A FloodFill eljárás használatakor szembe kell néznünk azzal a problémával, hogy ha egy kör túl közel kerül a kerethez, és ráadásul a keret is fekete színű, akkor a FloodFill a keretbe is belefestene. Most érthetjük meg annak a vékony üres keretnek a szerepét, amit a program tárgyalásának kezdetén említettem.

Stringtábla erőforrások használata

A harmadik típusú erőforrás, amivel ebben a fejezetben foglalkozunk, a stringtábla. A Delphi 3-ban a stringtábláknak sokkal nagyobb szerepe jutott, mint a fejlesztőkörnyezet korábbi verzióiban. Ahogy a 4. fejezetben bemutattuk, ha egy szöveg-konstansra van szükségünk egy programban, akkor azt a `const` rész helyett egy `resourcestring` részben deklarálhatjuk:

```
resourcestring
  Text1 = 'This is some text';
```

Ekkor a Delphi automatikusan az alkalmazás erőforrás-táblájában hoz létre új bejegyzést. Ezt a táblát a rendszer egy STR kiterjesztésű fájlban tárolja, amelyet fordításkor beépít a futtatható fájlba. Ez azt jelenti, hogy valahányszor a fenti Text1 szöveg-konstansra hivatkozunk, a Delphi automatikusan az erőforrásból fogja kiolvasni a stringet. Hamarosan részletesebben is megismerjük ezt.

Ennek a változtatásnak a következménye, a program adat és kódrészének újfajta elrendezése a memóriában, ami általában javít a programon, mert a rendszer nagyon hatékony módon tudja kezelni az erőforrásokat. A stringek a memóriában a kóddal együtt találhatóak, míg a stringtáblák különálló, felszabadítható memóriablokkokban, amelyeket az alkalmazás akkor tölt be, amikor szükség van rájuk (valójában a Delphi 3 technikai okokból vezette be ezt az újítást, hogy elkerülje a különböző csomagok ugyanolyan azonosítóval rendelkező erőforrásainak összeütközését, de végül ezt az új lehetőséget kaptuk, és kihasználhatjuk minden előnyét).

A memória-kezelésen kívül, még egy ok szól az erőforrások használata mellett: egy programot nemzetközivé tehetünk segítségükkel (például angol helyett magyar nyelvűvé alakíthatjuk). Egy hagyományos programot, ha más nyelvűvé akarunk tenni, akkor végig kell néznünk az összes forrásfájlt, meg kell keresnünk a beágyazott szövegeket, le kell fordítanunk azokat, és utána újra le kell fordítanunk a programot. Ellenben, ha egy stringtáblákat használó programot akarunk lefordítani, akkor csak az erőforrásokban található szövegeket kell

lecserélnünk, csak az erőforrás fájlokat kell újrafordítanunk (ezt rendkívül egyszerű megtenni), és végül az új erőforrásokat kell a lefordított EXE fájlhoz kötnünk.

Az alkalmazás más nyelvre való lefordítása a Delphi alatt is hasonlóan történik. A felhasználói felület lefordítása leginkább a komponensek feliratainak lefordítását jelenti. Ezt a form fájlok szöveges leírásában tehetjük meg legkönnyebben (amit úgy tudunk elérni, ha a DFM fájlokat a Delphi kódszerkesztő ablakában nyitjuk meg). Ezeken kívül a program valószínűleg meg akar jeleníteni olyan, a felhasználónak szóló üzeneteket, amelyeket nem akarunk mindig külön formon megjeleníteni.

Megjegyzés

A Borland és más cégek speciális eszközöket árulnak a programok nemzetközivé tételéhez. Kaphatók egyszerű és olcsó komponensek, amelyek adatbázis táblákat használnak, de beszerezhetőek összetett és drága rendszerek is. Ezek közül némelyek lehetővé teszik, hogy több különböző futtatható fájlt készítsünk, minden nyelvnek egyet, sőt, a nyelvet a program újraindítása nélkül is megváltoztathatjuk.

Egy egyszerű megoldási lehetőség, hogy az összes üzenetet stringtáblákban helyezzük el. Az így készült programokat ugyanolyan könnyen írhatjuk át más nyelvűre, mint azokat a hagyományos alkalmazásokat, amelyek a stringeket kizárólag Windows API hívásokkal érik el. Pontosan ezt láthatjuk a következő, StringT példaprogramban. Ennek a megközelítésnek alaposabb kihasználásához, és, hogy megértsük hogyan működnek a stringtáblák a felszín alatt, a program nem a Delphi 3 `resourcestring` lehetőségét veszi igénybe, hanem közvetlenül éri el a stringtáblákat. A StringT program formja két részre osztható, ezt a két részt két téglalap jelöli. Az egyikben egy nyomógomb, a másikban egy újabb nyomógomb és egy szerkesztő mező található (a hozzájuk rendelt címkéken kívül).

A példa legfontosabb rész az erőforrások forrásfájl (aminek kiterjesztése most is RC). Ez egy átlagos szövegfájl, amiben erőforrás utasítások szerepelnek. Ezeknek az utasításoknak a dokumentációját a Windows API súgófájljában találhatjuk meg. Az STABLE.RC fájl listája a következő:

```
STRINGTABLE
BEGIN
  1, "Hello"
  2, "String Table Test"
```

```
    11, "First message"
    12, "Second message"
...
    20, "Tenth message"
END
```

Egy stringtábla erőforrás definiálásához a fájlban szerepelnie kell a megfelelő (STRINGTABLE) utasításnak és egy listának, melyben azonosítók és a hozzájuk rendelt stringek szerepelnek. A vesszők használata nem kötelező. Felhívom a figyelmet arra, hogy a stringeknek a C nyelvnek megfelelő formátumban kell szerepelniük, tehát dupla idézőjeleket kell használnunk a Pascalban megszokott egyszeres aposztrófok helyett. Az erőforrás blokk elejét és végét jelző határolók lehetnek a Pascalnak megfelelőek (tehát a BEGIN és az END, nagybetűkkel), ahogy a példában is láthatjuk, de lehetnek a C-ben használt kapcsos zárójelek is: { és }.

Figyelem!

A stringek azonosításához tetszőleges számokat használhatunk, de érdemes végiggondolni, hogy a Delphi magukat a karakterláncokat használja, tehát lehetőleg ne szerepeljenek duplán bejegyzések. A Delphi általában nagyon nagy értékeket használ string azonosítóként, 60000 felettieket, tehát ha ez alatti értékeket használunk, az semmilyen problémát nem okozhat.

Ha ez a fájl elkészült, akkor le kell fordítanunk RES kiterjesztésű fájllá, mert csak ebben a formában tudjuk közvetlenül egy Delphi projektbe beépíteni. Ez minden reményünket meghaladóan egyszerű, hiszen a Delphi tartalmaz egy teljes erőforrás fordítót, a BRCC32-t, bár ez nem egy Windows alkalmazás, hanem parancssoros eszköz. Használatához indítsunk egy DOS-parancssor ablakot, váltsunk a projektet tartalmazó alkönyvtárba, és ott adjuk ki a következő parancsot:

```
BRCC32 -R STABLE.RC
```

Természetesen, meg kell adni a teljes elérési útvonalat is a BRCC32 fájlhoz, ami a Delphi\3.0\Bin alkönyvtárban található. Alapértelmezésben a BRCC32 azonnal megpróbálja az eredmény RES fájlt a megfelelő EXE fájlhoz kapcsolni. A -R paraméterrel adjuk meg a fordítónak, hogy az RC fájl lefordítása után hagyja el az összekapcsolási lépést. Hogyha elkészült a RES fájl, akkor a szokásos módon kapcsolhatjuk a futtatható fájlhoz:

```
{$R STABLE.RES}
```

Most már hozzáfoghatunk a valódi program megírásához. Az első stringet akkor akarjuk beolvasni és megjeleníteni, amikor a formon megnyomják a Hello gombot. A LoadString API függvényhívásnak van egy harmadik paramétere, amelyben a 0 karakterrel végződő stringet el kell helyezni. A függvény visszatérési értéke az erőforrásból beolvasott karakterek számát adja meg, illetve 0-t ha valamilyen hiba történt (például nem található a kívánt string). Ezért ellenőrizzük a visszatérési értéket:

```
procedure TForm1.HelloButtonClick(Sender: TObject);
var
  Text: array [0..255] of Char;
  N: Integer;
begin
  N := LoadString (HInstance, 1, Text, SizeOf (Text));
  if N > 0 then
    ShowMessage (StrPas (Text));
end;
```

Bár ez a módszer működőképes, mégis túlságosan bonyolult. Egy sokkal szebb megoldás a Delphi LoadStr függvényének használata, ami meghívja a Windows API függvényt, és az erőforrásból beolvasott stringet közvetlenül egy Pascal stringben helyezi el. A második verziója ugyanennek a metódusnak, ezek alapján, a következő:

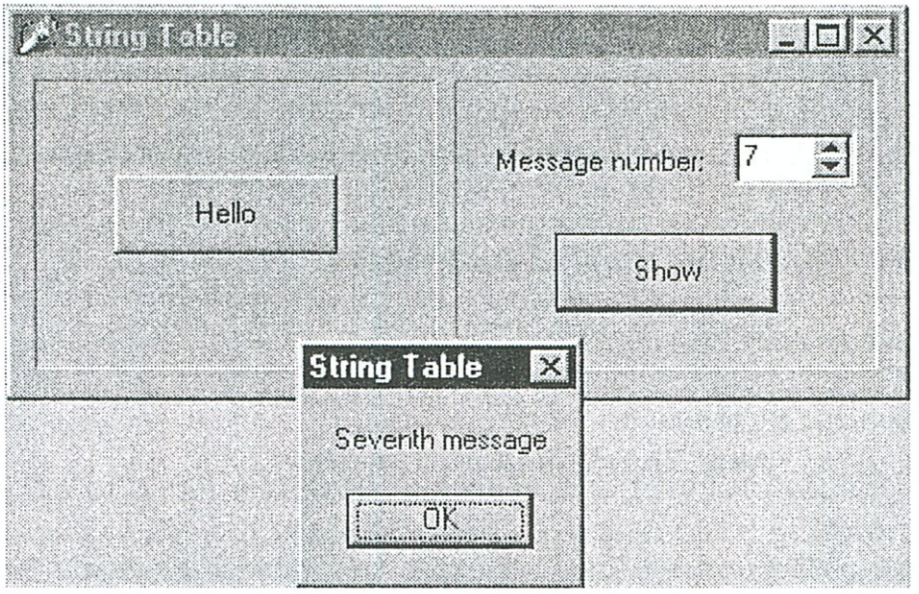
```
procedure TForm1.HelloButtonClick(Sender: TObject);
var
  StrMsg: string;
begin
  StrMsg := LoadStr (1);
  if StrMsg <> '' then ShowMessage (StrMsg);
end;
```

Még egy utasítás létezik a Delphiben, ami stringek betöltésére szolgál. Az FmtLoadStr eljárás a stringtábla szövegelemét az adatparaméterek formátumstringjeként használja fel. A LoadStr és a Format függvények funkcióját egyesíti ez az eljárás. A LoadStr eljárás újbóli meghívásával változtatjuk meg az ablak címét (és a lekicsinyített ablak címét) a második string felhasználásával:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  StrTitle: string;
begin
  StrTitle := LoadStr (2);
  if StrTitle <> '' then
  begin
    Caption := StrTitle;
    Application.Title := StrTitle;
  end;
end;
```

A Hello mellett a programnak van egy második nyomógombja, a Show, aminek egy egész kicsit bonyolultabb kódja van. Egy bizonyos string megjelenítése helyett, a ShowButtonClick metódus a szerkesztőmező aktuális értékét használja fel annak meghatározására, hogy melyik stringet használjuk a lehetséges 10-20 tartományból (ennek a metódusnak az eredményét láthatjuk a 27.11. ábrán):

```
procedure TForm1.ShowButtonClick(Sender: TObject);
var
  StrMsg: string;
begin
  StrMsg := LoadStr (10 + SpinEdit1.Value);
  if StrMsg <> '' then
    ShowMessage (StrMsg);
end;
```



27.11. ábra *A StringT példaprogram egy lehetséges kimenete a Show gomb megnyomása után*

Ez utóbbi eljárás a stringeknek egy újabb felhasználási területét is bemutatja. Mi mindössze 10-féle üzenetet jelenítünk meg, de pillanatok alatt új bejegyzéseket adhatunk a stringtáblához, megnövelhetjük a SpinEdit komponens MaxValue tulajdonságának értékét, és ezzel lehetővé tehetjük, hogy a program új stringeket jelenítsen meg, anélkül, hogy akár egyetlen sort is írnánk programunkban. Problémát jelenthet, hogy ha egy komponens valamely tulajdonságát (példánkban a MaxValue értékét) megváltoztatjuk, akkor újra kell fordítanunk a projektet. Ezen segít az a megoldás, hogy magának a tulajdonságnak az értékét is a stringtáblában tároljuk, és futás közben olvassuk be.

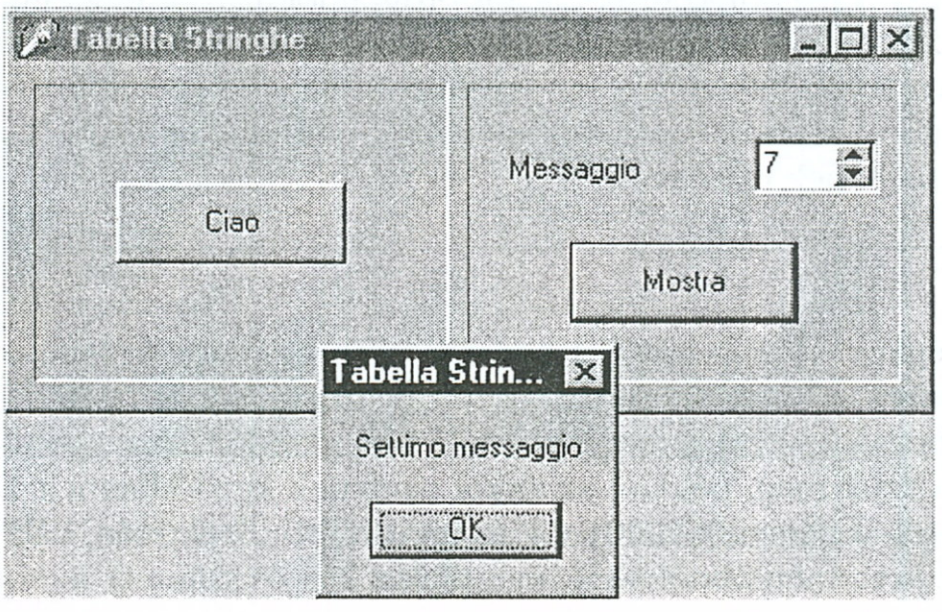
A stringek lefordítása idegen nyelvre

Az elkészült program tökéletesen működik, de igazi előnye az, hogy az erőforrásban tárolt stringek használatával egyszerűen megváltoztathatjuk az alkalmazás nyelvét. Csak az RC fájl nyelvét kell új nyelvre lefordítani (nem feltétlenül a programozónak), majd az új nyelvű fájlt kell az erőforrás fordítóval a futtatható fájlhoz kapcsolni. Ha az új fájlnak más a neve, mint az eredetinek, akkor az állományt beépítő fordítói direktívában kell kicserélni a nevet, és ezután újrafordítani a programot. Mindössze ennyit kell tenni, illetve még ennyit sem, hogyha a régi fájlnevet használjuk, vagy ha a külső erőforrás fordítót használjuk az állomány programhoz kötését.

A példában először lefordítottam a stringeket egy új nyelvre (olaszra). Ezek után az eredményül kapott STABIT.RC fájl így néz ki:

```
STRINGTABLE
BEGIN
  1, "Ciao"
  2, "Prova Tabella Stringhe"
  11, "Primo messaggio"
  12, "Secondo messaggio"
  ...
  20, "Decimo messaggio"
END
```

Ezután újrafordítottam a forrásfájlt és kicseréltem az erőforrásfájlt beépítő utasítást {$R STABIT.RES}. Ezek után ha újrafordítjuk és futtatjuk a programot, most a form címét és minden üzenetet olaszul jelenít meg. Ezt láthatjuk a 27.12. ábrán.



27.12. ábra *A StringT példaprogram olasz verziója*

A program még sincs teljesen lefordítva, a gombok feliratát is át kellene ültetni az új nyelvre. Ezt a form szöveges leírásában tehetjük meg, amit úgy érhetünk el, hogy a Delphi kód-szerkesztőjében nyitjuk meg. Itt ki kell cserélnünk az összes stringet: a feliratokat és a listaelemeket. Ne fordítsuk le viszont a belső azonosítókat, mint amilyenek például a Name tulajdonságok.

Én elvégeztem ezeket a változtatásokat, és a formot leíró új fájlt az STRFIT.DFM néven mentettem el. Ezt a {$R *.DFM} utasítás lecserélésével olvashatjuk be, az új direktíva: {$R STRFIT.DFM}. Most már minden felirat és üzenet olaszul jelenik meg, annak ellenére, hogy ugyanaz maradt a program forráskódja. Ha még egyszerűbbé akarjuk tenni az új verziók lefordítását ugyanazon forrásfájl felhasználásával, használhatjuk a feltételes fordítást eredményező fordítói direktívákat:

```
{$IFNDEF ITALIAN}
  {$R *.DFM}
  {$R STABLE.RES}
{$ELSE}
  {$R STRFIT.DFM}
  {$R STABLEIT.RES}
{$ENDIF}
```

Egy komolyabb programban, ezeken túl, a Delphi rendszer üzeneteit is le kell fordítanunk, például azokat, amelyek a VCL forrásfájlokban találhatóak (mint amilyen a CONST.RC vagy a DBCONSTS.RC). Ezzel válik teljessé a program lefordítása.

Megjegyzés

A Borland cég a Delphi Translation Kit programot ajánlja, egy egyszerű és gazdaságos csomagot, amely egy varázslót épít be a Delphi környezetbe. Kiválaszthatjuk a rendszerhibák nyelvét, az alapértelmezett gombok feliratát, az alapértelmezett párbeszédablakok és más hasonló elemek nyelvét is.

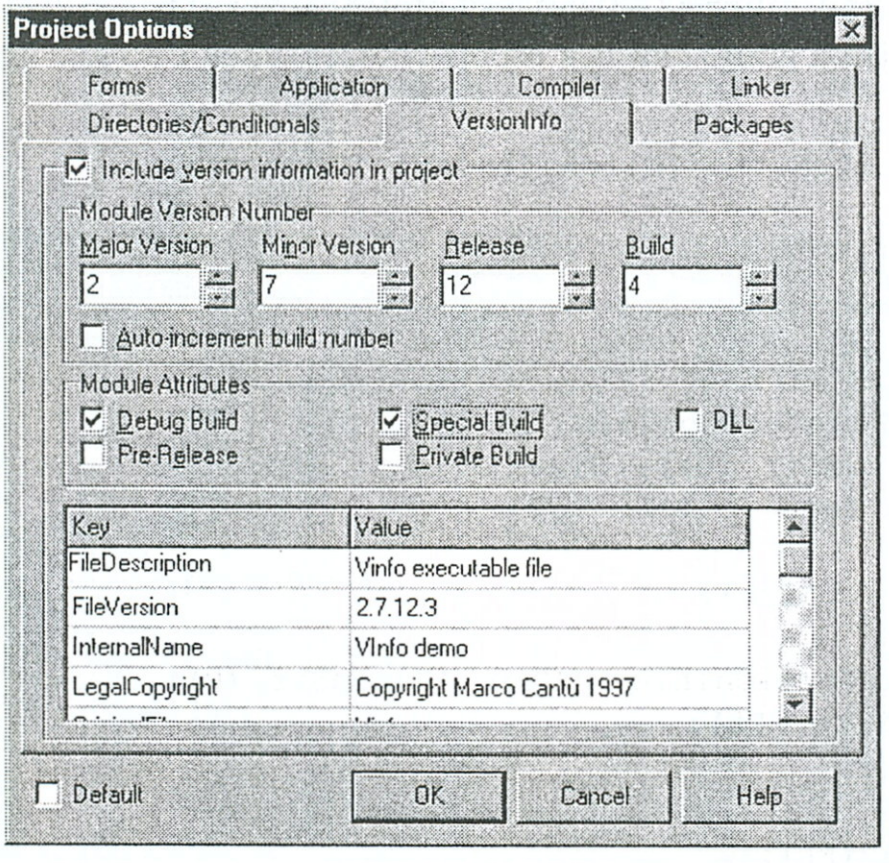
Létezik egy másik megközelítési módja is a nemzetközi programok készítésének. Ezzel a módszerrel először elkészítünk egy-egy string-csoportot mindegyik nyelvhez, méghozzá úgy, hogy mindegyikben a karakterláncokat ugyanolyan stílusban számozzuk meg, például az angol nyelvűek kezdődhetnek 1-től, a franciák 10001-től, a németek 20001-től, és így tovább. Ezek után, valahányszor egy stringet szeretnénk beolvasni (mondjuk a 235-ös számút), akkor a következőt írhatjuk:

```
MyText := LoadStr (nLanguage * 10000 + 235);
```

Ahol az nLanguage az éppen beállított nyelvnek megfelelő kód (0 az angol,1 a francia,2 a német, és így tovább). Ezek után a felhasználó futás közben, egy párbeszédablakban állíthatja be a kívánt nyelvet, a fenti kód megváltoztatásával. Még egy utolsó megjegyzés, amit érdemes programjainkban fontolóra venni: ha a stringekre számmal hivatkozunk, akkor azzal nemcsak nehezen olvashatóvá tesszük programunkat, hanem könnyel el is téveszthetjük a számokat. Ezért érdemes konstanst definiálni mindegyik karakterlánc azonosítójához, és ezeket a konstansokat használni a számok helyett.

Verziószámok kezelése

Az utolsó típusú erőforrás, amivel ebben a fejezetben részletesen foglalkozunk a verziószám. Már a Delphi 2-ben is használhattuk ezt az erőforrást, azonban a Delphi 3 kifejezetten a verziószámot kezelő módszerekkel támogatja használatát. Egyszerűen meg kell nyitnunk a projekt beállítások párbeszédablakot, átváltanunk a VersionInfo lapra, és máris beírhatjuk a következő információkat: az erőforrás számát, az alkalmazás nevét, másolási jogszabályokat és még sok egyéb információt. A 27.13. ábrán látható ez a dialógusablak.



27.13. ábra *A Project Options párbeszédablak VersionInfo lapja*

A verziószámot a DLL-ek és OLE szerverek (köztük az ActiveX vezérlők) használják fel. Telepítő programok ezek alapján tudják eldönteni, hogy az adott DLL-ből a legutóbbi verzió áll-e rendelkezésre. E nélkül fennállna annak veszélye, hogy egy régebbi DLL-lel vagy vezérlővel helyettesítünk egy újabbat. Különösen fontos ismernünk a DLL verziószámát és a hozzá tartozó adatokat, ha ki akarjuk használni a telepítő program lehetőségeit (vagy hogyha saját telepítőt írunk, és össze akarjuk hasonlítani a rendszerben már meglévő fájlt a sajátunkkal).

Mivel a verziószám mellett tárolt adatokról pontos leírást kaphatunk a Microsoft dokumentációban, azt itt most nem részletezem. Ehelyett inkább nézzünk egy egyszerű példát a verziószám használatára egy futtatható programban. Csak beállítunk néhány dolgot a verziószám erőforrásban, és egy olyan programot készítünk, amely a formon lévő, Read Version Info gomb megnyomásának hatására egy Memo komponensben megjelenít néhány információt. A probléma sajnos az, hogy az API használata a verziószám adatainak kinyerésére nem egyszerű.

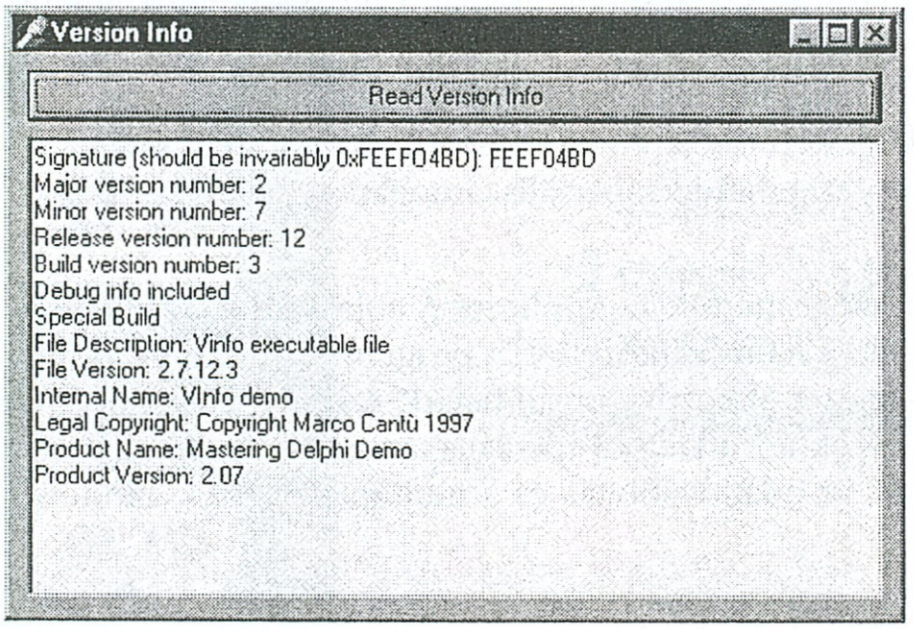
A használható API függvény neve GetFileVersionInfo. Paraméterei közt meg kell adni a fájlnevet, egy pointert, ami oda mutat a memóriában, ahova az adatokat elhelyezheti az eljárás, és ennek a memória-területnek a méretét. A megfelelő méretű terület lefoglalásához a GetFileVersionInfoSize hívást használhatja a program. Ezek alapján a gomb OnClick eseménykezelőjének első fele így néz ki:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Size, Size2: DWord;
  Pt, Pt2: Pointer;
begin
  Memo1.Lines.Clear;
  Size := GetFileVersionInfoSize (
    PChar (ParamStr (0)), Size2);
  if Size > 0 then
  begin
    GetMem (Pt, Size);
    try
        GetFileVersionInfo (PChar (ParamStr (0)),
          0, Size, Pt);
        ...
    finally
      FreeMem (Pt);
    end;
  end;
end;
```

A fenti kódrészlet utolsó részében, a finally blokkban felszabadítjuk a memóriát. Közben a program a Pt pointert használja a verziószám-adatok elérésére. Az adatok eléréséhez a VerQueryValue API függvényt használjuk, amelynek meg kell adni paraméterként az előbbi pointert, egy stringet, amelyben az szerepel, hogy milyen információra van szükségünk, és egy újabb pointert, amelyet az eljárás ráállít arra a területre, ahol a kívánt string vagy adatstruktúra megtalálható.

Jelenlegi példánk első része a fájl információk állandó részét éri el: néhány kapcsolót és számokat, amelyeket a TVSFixedFileInfo struktúra definiál. Ezek közül néhányat az alábbi programrészlet ér el (a melléklet CD-n bővebb változat szerepel):

```
// show the fixed information
VerQueryValue (Pt, '\', Pt2, Size2);
with TVSFixedFileInfo (Pt2^) do
begin
  Memo1.Lines.Add (
    'Signature (should be invariably 0xFEEF04BD): '
    + IntToHex (dwSignature, 16));
  Memo1.Lines.Add ('Major version number: ' +
    IntToStr (HiWord (dwFileVersionMS)));
  if (dwFileFlagsMask and dwFileFlags
      and VS_FF_DEBUG) <> 0 then
    Memo1.Lines.Add ('Debug info included');
```



27.14. ábra *A VInfo példaprogram kimenete: Egy program fájl-információit mutatja meg*

Az eljárás második részben olyan stringeket olvasunk ki, amelyek a program verziószám erőforrásában szerepelnek. Mindegyik karakterláncot külön érhetjük el, megint a VerQueryValue API hívással. Íme néhány ezek közül:

```
VerQueryValue(Pt,
  '\StringFileInfo\040904E4\FileDescription', Pt2,
Size2);
Memo1.Lines.Add ('File Description: ' + PChar (pt2));
VerQueryValue(Pt,
  '\StringFileInfo\040904E4\FileVersion', Pt2, Size2);
Memo1.Lines.Add ('File Version: ' + PChar (pt2));
```

Ennek a programnak az eredménye a 27.14. ábrán látható. A látható értékek nem túl beszédesek, a verziószámokat és egyéb értékeket véletlenszerűen írtam be. Bizonyos alkalmazások ezeket az adatokat arra használják, hogy a Névjegy ablakban megjelenítsék a program nevét és verziószámát.

Hogyan tovább?

Ebben a fejezetben megvizsgáltuk a Windows erőforrások szerepét, definiálását és használatát a hagyományos alkalmazásokban, és láthattunk példaprogramokat Delphiben való használatukra. Mint sok más grafikus fejlesztői környezetben, a Delphiben is lehetőségünk nyílik a legtöbb elem grafikus felhasználására, és ezzel egyéni megoldásokat használhatunk a Windowsban megszokott megoldások helyett, amelyek az erőforrásokra építenek. Időnként mégis szükségünk van az erőforrások használatára. A korábbi fejezetekben bitképeket töltöttünk be az alkalmazás erőforrásai közül, ebben a fejezetben pedig láthattuk az ikonok és a kurzorok betöltésének módját. Ugyancsak áttekintettük egy nem-grafikus erőforrástípus kezelését, a stringtáblákat, melyek ugyanolyan hasznosak a Delphiben, mint más, hagyományosabb programozási környezetekben. A stringtáblák kezelése annyira egyszerű szöveges fájlok segítségével, hogy a Delphi környezet minden speciális támogatása nélkül is elboldogulhatunk velük.

A könyv eddigi részében megismertük az Application objektumot, az alkalmazások általános használatát, a memóriakezelést, a programtesztelést és az erőforrásokat. A következő fejezettől kezdve olyan haladó szintű fogalmakkal fogunk megismerkedni, melyekhez a Delphi speciális támogatást nyújt. Ilyenek a fájlkezelés, a nyomtatás, az adattovábbítás és a multimédia. Ezek az utolsó négy fejezet témái.

28. FEJEZET

Nyomtatás Delphi alkalmazásokból

E fejezet tartalma:

- Az egész form kinyomtatása
- A Printer objektum használata
- Szöveg nyomtatása
- Adatbázisrekordok és -táblák nyomtatása
- A QuickReport jelentések készítése
- ReportSmith jelentések készítése

Nyomtatás Delphi alkalmazásokból

Bár a Delphi programozás számos területét, köztük néhány összetettebbet is bejártuk, néhány fontos dologról még nem esett szó. Ezek közé tartozik a nyomtatás, amiről ebben a fejezetben fogunk beszélni. Bizonyos nyomtatási szolgáltatásai még az olyan egyszerűbb mindennap használatos programoknak is vannak, amilyeneket az előző fejezetekben készítettünk. E fejezet példái közül sok korábbi példák kibővített változata, melyek immár ezekkel a lehetőségekkel is rendelkeznek.

Ez a fejezet két, a nyomtatással kapcsolatos témát tárgyal; az első rész a formok nyomtatásával foglalkozik, beleértve a szöveg és a grafika nyomtatását is, a második rész a jelentések nyomtatását mutatja be. A Delphi mindkét módszert támogatja, így létező programjainkat könnyedén kibővíthetjük ezekkel a szolgáltatásokkal.

Az egész form kinyomtatása

Egy Delphi formot futásidőben kinyomtatni nagyon egyszerű, mert a TForm osztálynak van Print (nyomtatás) metódusa, ami az egész kliens területet kinyomtatja, a rajta lévő vizuális komponensekkel együtt. A nyomtatáshoz elég csak ezt a metódust meghívnunk. Például a formunkra tehetünk Print (nyomtatás) gombot, amihez a következő kód tartozhat:

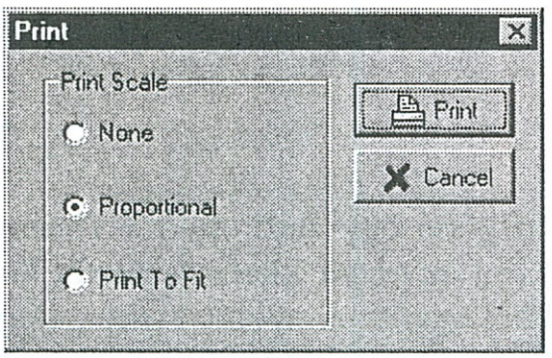
```
procedure TForm1.PrintButtonClick (Sender: TObject);
begin
  Print;
end;
```

Ennek a programnak a hatására az aktuális nyomtatón megjelenik a form kliens területének – vagyis a formnak a keret és a címsor nélkül – képe. A nyomtatott kép méretének meghatározására a Print metódus meghívása előtt használhatjuk a form PrintScale (nyomtatási méretezés) tulajdonságát. A tulajdonság alapértelmezett értéke poProportional. Ez azt jelenti, hogy a nyomtatás a képernyő képhez képest arányos lesz, a form PixelPerInch (képpont/hüvelyk) tulajdonságát a képernyő méretéhez viszonyítva, és a nyomtató DPI (pont/hüvelyk) tulajdonságát a papír méretéhez viszonyítva. Például ha a form fél képernyő széles, akkor a nyomtatott változat a nyomtatott oldal felét fogja elfoglalni.

Ennek a tulajdonságnak még két másik értéke lehet, a poNone, és a poPrintToFit. A poNone használatakor a kép nem méreteződik át így a nyomtatók magas hüvelykenkénti pontsűrűsége miatt a kapott ábra elég kicsi lehet. Ha a poPrintToFit paramétert használjuk, akkor a kép megnyúlik, akkorára, hogy az egész oldalt elfoglalja, de megtartja az eredeti szélesség magasság arányt. A form PrintScale tulajdonságánál a poPrintToFit használatával általában sokkal nagyobb képeket kapunk, de mivel a form képe megnyúlik, ezért a nyomtatási minőség a kép méretének növelésével romlik.

Saját nyomtatási párbeszédablak

Bár a Delphiben van olyan komponens, ami a szabványos Windows nyomtatási párbeszédablakot jeleníti meg, az egyszerűség kedvéért az első példánkban mégsem használjuk ezt. Ehelyett egy egyszerű párbeszédablakot készítettem, három rádiógombbal (valójában egy három elemű rádiócsoporttal), amikkel a PrintScale tulajdonság lehetséges értékei közül választhatunk. Ezt a nyomtatási párbeszédablakot a tizenkettedik fejezetből ismert ComboBar példába illesztettük, amit most PrnFormnak hívunk. A 28.1. ábrán látható a főablak, és a nyomtatási párbeszédablak. A File menühöz hozzáadtam egy Print parancsot, és a form kliens felületét eltakaró címke háttérszínét fehérre állítottam.



28.1. ábra *A PrnForm példa futás közben*

Ennek a példának egyszerű formja van, amin sokféleképpen változtathatjuk a címke betűtípusát, és a hatást mind képernyőn, mind papíron megnézhetjük. Az egyetlen probléma az, hogy a nyomtatott kép nem csak a címke szövegét tartalmazza, hanem az eszköztár képét is. Ha ezt el akarjuk kerülni, akkor elrejthetjük az eszköztárat az Toolbar menü Visible parancsával, vagy az eszköztár gyorsmenüjével. Ily módon a programot felhasználhatjuk a címke szövegének kinyomtatására. Ha most még hozzáadunk egy szerkesztőmezőt, vagy egy Memo mezőt, amivel a címke szövegét megváltoztathatjuk, akkor bármit kinyomtathatunk. Miért is vennénk drága szövegszerkesztőket?

A példa kódja nagyon egyszerű, amikor a felhasználó kiadja a Print parancsot, a saját párbeszédablakunk jelenik meg. Ha a felhasználó a Print gombot nyomja meg (aminek mrOk érték szerepel a ModalResult értékében), az aktuálisan kiválasztott átméretezési opciót beállítjuk a PrintScale tulajdonságba, és a formot kinyomtatjuk:

```
procedure TToolbarForm.Print1Click(Sender: TObject);
begin
  if PrintDlg.ShowModal = mrOk then
  begin
    case PrintDlg.RadioGroup1.ItemIndex of
```

```
          0: PrintScale := poNone;
          1: PrintScale := poProportional;
          2: PrintScale := poPrintToFit;
        end;
      Print;
    end;
  end;
```

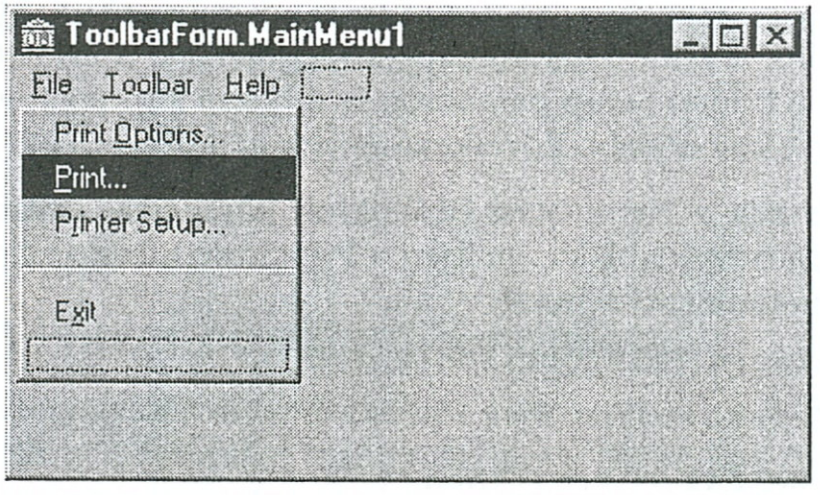
Maga a párbeszédablak nem is igényel több kódot, mint amit a Delphi az osztálydefinícióhoz legenerál. Ha tovább akarnánk menni, akkor még hozzátehetnénk egy panelt, ami a nyomtatási képet mutatná. Létezik ugyanis a `TForm` osztálynak egy GetFormImage metódusa, ami bittérképként visszaadja a form képét. Ez ugyanaz a kép, ami a Print metódus kódjának meghívásakor a nyomtatóra kerül.

Ami nem egyszerű, az a nyomtatási átméretezés szimulálása a nyomtatott oldalnak megfelelő területen. A fejezetben még lesz egy olyan példa (PrintBmp), amiben lesz ilyen nyomtatási kép párbeszédablak. Az ottani kódot felhasználhatjuk a Combo2 példa javítására is.

A szabványos nyomtatás párbeszédablakok

A PrnForm példában használt saját Nyomtatási párbeszédablak helyett használhatjuk, a Delphibe beépített két szabványos Windows párbeszédablak egyikét, a PrintDialog, vagy a PrinterSetupDialog komponenseket.

Ez az a megközelítés, amit a következő, PrnForm2 nevű példában használunk. Ebben a példában az előző változat párbeszédablakát Nyomtatási Opciók párbeszédablakként használjuk, emellett szerepelnek még a szabványos Print (nyomtatás) és Printer Setup (nyomtató beállítások) ablakok is. A változásokat tükrözendő, átalakítottuk a Fájl menüt, ahogy az a 28.2. ábrán látható.



28.2. ábra *A PrnForm2 példa Állomány menüje a Delphi menütervezőjében*

A párbeszédablakokat szimbolizáló két komponensnek nincsenek különleges tulajdonságai, és egészen egyszerűen használhatóak. A nyomtató beállításához elég, ha a PrinterSetupDialog1 komponens Execute metódusát meghívjuk:

```
procedure TToolbarForm.PrinterSetup1Click(
  Sender: TObject);
begin
  PrinterSetupDialog1.Execute;
end;
```

Ettől eltérően, a PrintDialog Execute metódusának visszatérési értékére is figyelnünk kell, mert innen tudhatjuk meg, hogy a felhasználó tényleg akar-e nyomtatni. Szintén szerepel még egy kódrészlet, ami a Print párbeszédablak egyetlen, mindig látható beállítását kezeli, ez pedig a másolatok száma. A PrnForm2 példa nem nyomtatja ki magától az összes másolatot, hanem az első befejezése leellenőrzi, hogy kell-e több, és ha igen, akkor megkérdezi a felhasználót, hogy tényleg szeretné-e a többit is:

```
procedure TToolbarForm.Print1Click(Sender: TObject);
var
  I: Integer;
begin
  if PrintDialog1.Execute then
  begin
    Print;
    if (PrintDialog1.Copies > 1) and (MessageDlg (
        'Do you actually want to print ' +
        IntToStr (PrintDialog1.Copies - 1) +
        ' more copies?', mtConfirmation,
        [mbYes, mbNo], 0) = mrYes) then
      for I := 1 to PrintDialog1.Copies - 1 do
        Print;
  end;
end;
```

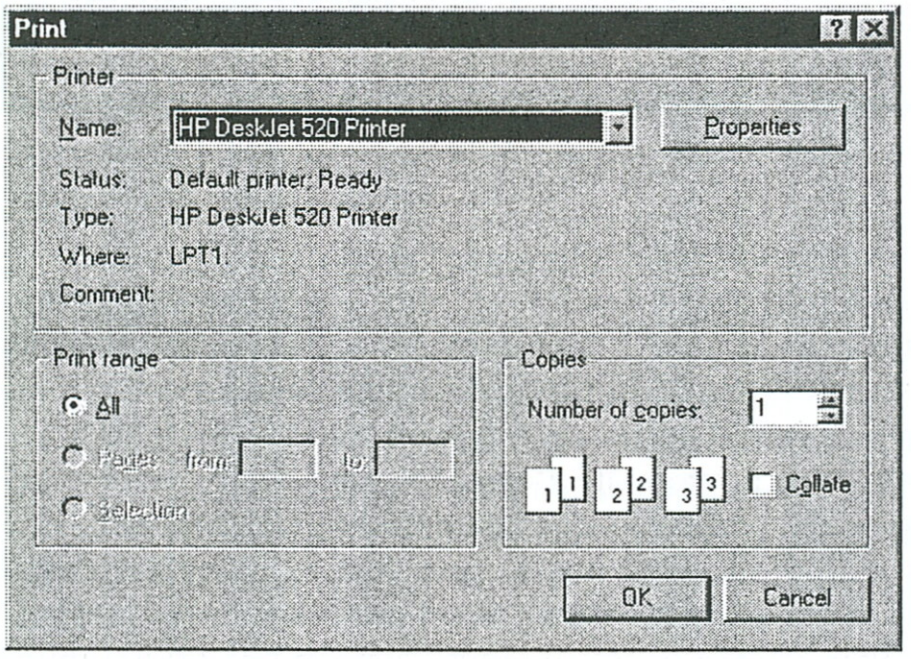
Amint az a 28.3. ábrán látható, a Print párbeszédablak meghívhatja a Printer Setup párbeszédablakot, így a Printer Setup parancsot, és a hozzá kapcsolatos komponenst is elhagyhatnánk különösebb következmények nélkül.

Amikor a felhasználó kiválasztja a Nyomtatási beállítások parancsot, akkor megjelenik az előző példa párbeszédablaka, amiben a PrintScale tulajdonság van kiválasztva, de nem nyomtatunk ki semmit:

```
procedure TToolbarForm.PrintOptions1Click(
  Sender: TObject);
begin
  if PrintDlg.ShowModal = mrOk then
  begin
    case PrintDlg.RadioGroup1.ItemIndex of
```

```
            0: PrintScale := poNone;
            1: PrintScale := poProportional;
            2: PrintScale := poPrintToFit;
          end;
          if PrintDlg.PrintRequest then
            Print1Click (self);
        end;
      end;
```

Ez a kód nem hívja meg a Print metódust, de elindítja a Print1Click metódust. A párbeszédablakunknak most három gombja van, az OK, a Cancel, és a Print. Az OK, és a Cancel a már megszokott módon működik, a Print gombnak a ModalResult tulajdonsága mrOK, de az OnClick eseménykezelője a form PrintRequest publikus mezőjét igazra állítja. Emiatt, a gomb megnyomásakor az ablak bezáródik, és a Print párbeszédablak jelenik meg.



28.3. ábra *Szabványos Windows 95 Nyomtatási párbeszédablak a PrnForm2 példában*

A Printer objektum használata

A legegyszerűbb műveletek – mint amilyen az egész form kinyomtatása – kivételével a nyomtató manipulálására a Delphiben definiált Printer globális változót használjuk. A valóságban a Printer egy globális függvény neve, ami egy, a Printers egységben definiált TPrinter osztályba tartozó objektumot ad vissza.

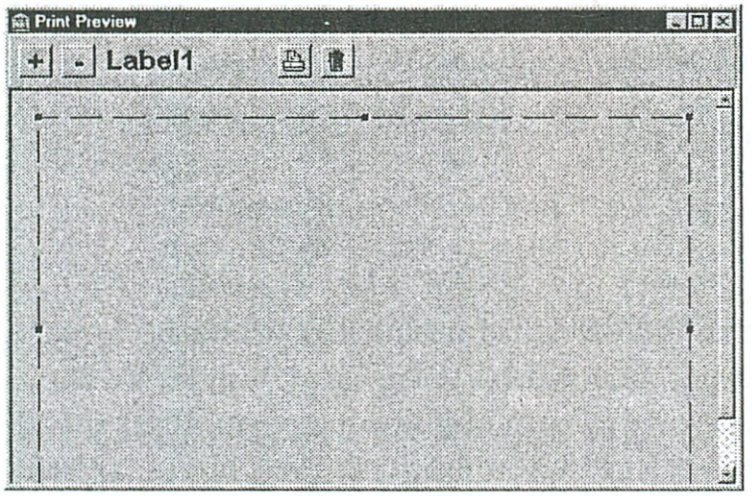
Megjegyzés

A 16 bites Delphiben a Printer egy, a TPrinter osztályba tartozó globális objektum neve volt. A 32bites Delphiben még mindig létezik TPrinter osztályú globális objektum, de ez most már a nyomtató egységhez kapcsolódik. Hogy ezt az globális objektumot elérjük, a Printer függvényt használhatjuk. Ez az új kódot kompatibilissé teszi a régiekkel. Például még mindig használhatjuk a `Printer.Canvas.Font` kifejezést a jelenlegi szöveg megváltoztatására. Ezt a változtatás azért következett be, mert a Delphinek már van egy SetPrinter (nyomtató beállítás) függvénye, amivel ezt a globális objektumot is megváltoztathatjuk, így több, különböző beállítású nyomtató objektumunk is lehet, és ezek egyikét szükség szerint hozzárendelhetjük az aktuális nyomtató objektumhoz. Ez hasznos lehet még, ha egy objektumot a TPrinter osztály leszármazottjaként szeretnénk létrehozni.

A Printer függvény által visszaadott objektum segítségével olyan, nyomtatással kapcsolatos globális tulajdonságokat érhetünk el, mint a telepített meghajtóprogramok, és betűtípusok listája, de a legfontosabb tulajdonsága mégis a vászon (Canvas). A nyomtató vásznát ugyanúgy használhatjuk, mint a form esetében tettük; írhatunk rá, rajzolhatunk, és még sok minden mást is. A vászon használatához hívjuk meg a nyomtató BeginDoc metódusát, ezzel jelezzük a nyomtatási munka elejét, ezután a Canvas metódusaival készítsük el a képet, majd hívjuk meg az EndDoc metódust, hogy a végeredmény a nyomtatóra kerüljön. Ezen kívül használhatjuk még az Abort metódust, hogy megszakítsuk a nyomtatási munkát, vagy a NewPage metódust, ami az eddigi munkánkat a nyomtatóra küldi, és új oldalt kezd.

Grafikák nyomtatási képe

Első példánkban a globális nyomtató objektum segítségével képeket fogunk nyomtatni. Ez voltaképpen a tizennegyedik fejezetben bemutatott TabOnly példa kibővítése. Az a példa TabControl komponens segítségével mutatott meg képsorozatokat. Ahogy az a 28.4. ábrán is látható, a PrintBmp példának van egy nyomtatási kép formja, egy eszköztárral, amin négy gomb van, és egy ScrollBox komponense, ami a kép komponenst tárolja. Ha a kép nagyobb mint a form, akkor a ScrollBox komponensben az eszköztár módosítása nélkül görgethetjük.



28.4. ábra *A PrintBmp példa nyomtatási kép formja tervezés közben*

Ez a nyomtatási kép párbeszédablak a File menü Print pontjára jelenik meg, és lehetővé teszi, hogy a keletkező bittérkép méretét összevethessük a lap méretével, amit a kép komponens szimbolizál, valamint, ha szükséges, átméretezhessük a képet.

Megjegyzés

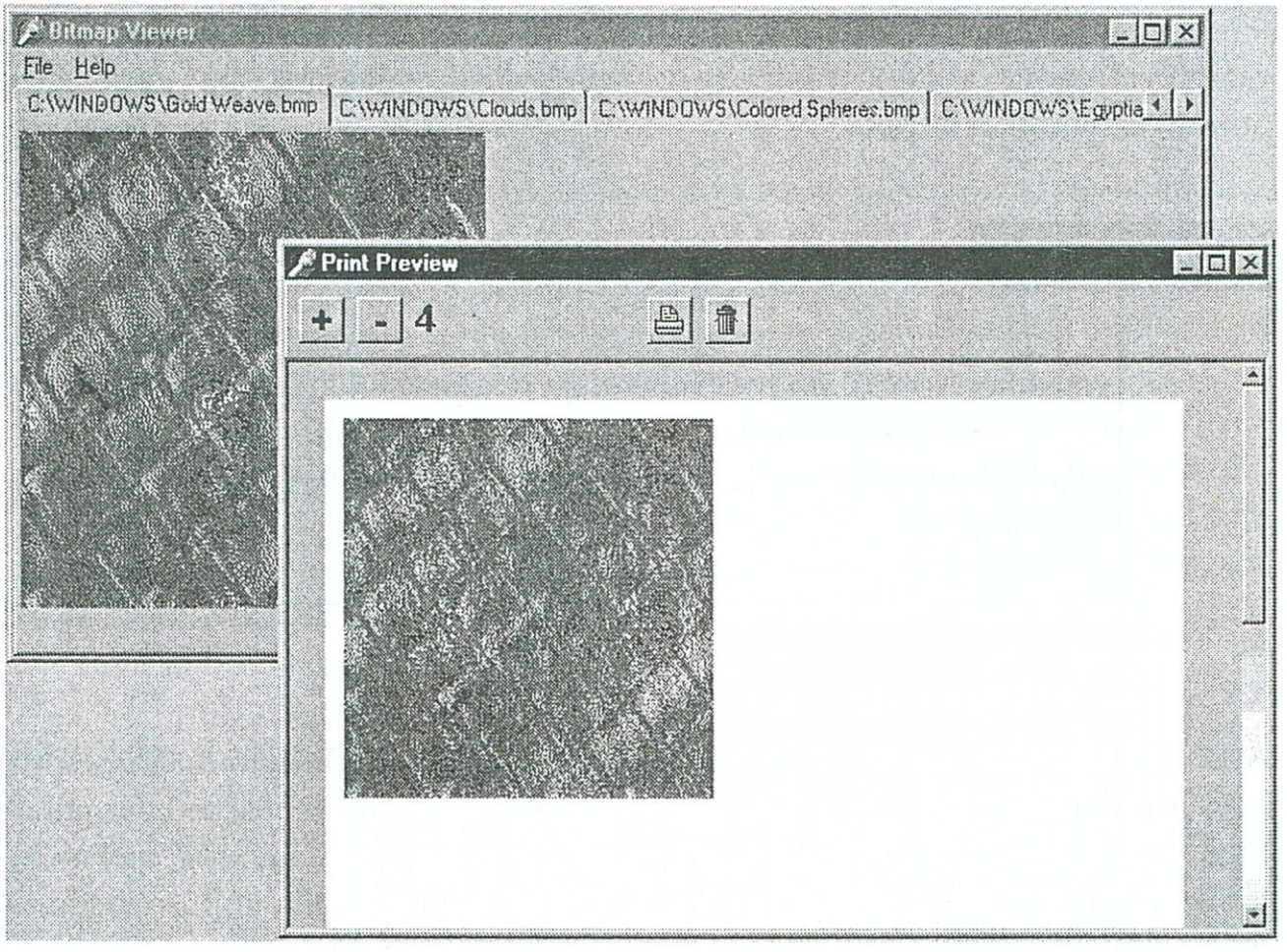
A kép méretének megváltoztatása mind a nyomtatott, mind az előzetes kép formában megváltoztatja a képernyő képet. A képet azért kell átméretezni nyomtatás előtt, mert a megszokott hüvelykenkénti pixel arány figyelembevételével a megjelenő ábra nagyon kicsi szokott lenni.

A kód a TCanvas osztály StretchDraw (nyújtott rajzolás) metódusán alapul, amit a nyomtatási kép, és a tényleges eredménykép elkészítésére használhatunk. A StretchDraw metódusnak két paramétere van: egy négyszög, ami a célterületet jelzi, és egy grafikus objektum, a forrásként szolgáló kép. A végeredmény egy, a célterület méretére nyújtott kép. Nézzük a kódot. A főform a Nyomtatás parancsra a Preview (Nyomtatási kép) form megnyitásával válaszol:

```
procedure TForm1.Print1Click(Sender: TObject);
begin
  {double-check whether an image is selected}
  if Image1.Picture.Graphic <> nil then
  begin
```

```
        {set a default scale, and start the preview}
        PreviewForm.Scale := 2;
        PreviewForm.SetPage;
        PreviewForm.DrawPreview;
        PreviewForm.ShowModal;
    end;
end;
```

Az első vizsgálat elhagyható, hiszen a Nyomtatás menüpont addig nem aktív, ameddig nincs kiválasztott kép, de ez biztosabbá teszi, hogy legyen kiválasztott kép. Ez a kódrészlet beállítja a PreviewForm objektum egy publikus mezőjét (Scale – méretezés), és meghívja a form két metódusát (SetPage – lapbeállítás és DrawPreview – nyomtatási kép rajzolása), végül pedig megjeleníti modális formként. A 28.5. ábrán láthatjuk a Print Preview formot működés közben.



28.5. ábra *A PrintBmp példa Print Preview formja, a főformmal a háttérben*

A SetPage metódus állítja be a Print Preview form kép komponensének méretét, a nyomtatott lap méretének felhasználásával:

```
procedure TPreviewForm.SetPage;
begin
  Image1.Width := Printer.PageWidth div 5;
  Image1.Height := Printer.PageHeight div 5;
  {output the scale to the toolbar}
  Label1.Caption := IntToStr (Scale);
end;
```

Az oldal méretét elosztottam öttel, hogy a képernyő egy elfogadható területén elférjen; esetleg felmerülhet, hogy a beépített érték helyett használhatnánk paramétert, ezzel nagyíthatóvá tehetnénk a nyomtatási képet is. Itt azonban zavarónak bizonyult, hogy külön gomb van arra, hogy a nyomtatott képet nagyítsuk, és arra, hogy a nyomtatási kép előzetesét növeljük, ezért itt most kimaradt a nagyítási lehetőség.

A nyomtatási kép form kódjának magja a DrawPreview metódus, ami három részből áll. Az elején kiszámítjuk a célnégyszöget, egy tíz pixeles margó meghagyásával, átméretezzük a képet, egy rögzített, öt értékű nagyítási aránnyal, ahogy az a következő listán is látható. A második lépésben kitöröljük az esetleg ottmaradt régi képet, azzal, hogy egy téglalapot rajzolunk rá. A harmadik lépés meghívja a vászon StretchDraw metódusát az előzőleg kiszámított téglalappal, és a főform kép komponensével (Form1.Image1.Picture.Graphic). Hogy ezeket az információkat elérhessük, a kód implementációs részének `uses` kifejezésébe be kell tennünk a TForm1 osztályt deklaráló Viewer egységet. Íme a metódus kódja:

```
procedure TPreviewForm.DrawPreview;
var
  Rect: TRect;
begin
  {compute the rectangle for the bitmap preview}
  Rect.Top := 10;
  Rect.Left := 10;
  Rect.Right := 10 +
    (Form1.Image1.Picture.Graphic.Width * Scale) div 5;
  Rect.Bottom := 10 +
   (Form1.Image1.Picture.Graphic.Height * Scale) div 5;
  {remove the current image}
  Image1.Canvas.Pen.Mode := pmWhite;
  Image1.Canvas.Rectangle (0, 0,
    Image1.Width, Image1.Height);
  {stretch the bitmap into the rectangle}
  Image1.Canvas.StretchDraw (Rect,
    Form1.Image1.Picture.Graphic);
end;
```

Ez a kód csak a form inicializálását végzi. Az indító kódot azért osztottam két részre, mert a DrawPreview-t még máshonnan is meghívjuk. Amikor az inicializálás megtörtént, és a modális form látható, a felhasználó rákattinthat a négy eszköztár gomb egyikére, hogy átméretezze, kinyomtassa, vagy átugorja a képet.

A két átméretező metódus nagyon egyszerű, mert csak az új méretarányt állítják be, és a kép frissítéséhez meghívják a DrawPreview eljárást:

```
procedure TPreviewForm.ScalePlusButtonClick(
  Sender: TObject);
begin
  Scale := Scale * 2;
  Label1.Caption := IntToStr (Scale);
  DrawPreview;
end;
```

A PrintButtonClick metódus majdnem megegyezik a DrawPreview-val, csak annyi az eltérés, hogy a cél téglalapot nem nagyítjuk ötszörösére, és hogy a kép a nyomtatóra kerül egy új dokumentumban (új oldalon):

```
procedure TPreviewForm.PrintButtonClick(Sender:TObject);
var
  Rect: TRect;
begin
  {compute the rectangle for the printer}
  Rect.Top := 10;
  Rect.Left := 10;
  Rect.Right := 10 +
    (Form1.Image1.Picture.Graphic.Width * Scale);
  Rect.Bottom := 10 +
    (Form1.Image1.Picture.Graphic.Height * Scale);
  {print the bitmap}
  Printer.BeginDoc;
  try
    Printer.Canvas.StretchDraw (Rect,
      Form1.Image1.Picture.Graphic);
    Printer.EndDoc;
  except
    Printer.AbortDoc;
    raise;
  end;
end;
```

A nyomtatási kód megosztása

Az egész form kinyomtatása, vagy egy létező kép használata helyett a nyomtató vásznára ugyanúgy rajzolhatunk, mintha az egy form, vagy más komponens felülete lenne. Mivel a nyomtató vásznán ugyanazokat a metódusokat

használhatjuk, mint bármely másik vásznon, ezért készíthetünk programokat két rajzoló metódussal, egyik a képernyőre, a másik a nyomtatóra rajzol, ugyanazzal a kóddal, vagy, ami még jobb, ugyanazt a rajzoló metódust használhatjuk mind a két felületen.

Ennek bemutatására a tizedik fejezet Shapes4 példáját újítottuk fel, ami alakzat leírások listáját tárolta a memóriában, majd ezeknek a leírásokat használta az OnPaint metódusban. Az új változatot Shape6-nak hívjuk, mert a Shape5 már a tizenegyedik fejezetben szerepel. A fejezet többi példájához hasonlóan, a form egyetlen újdonsága a Fájl menü Nyomtatás pontja.

A példa érdekessége, hogy a FormPaint rész kódját áttettük egy másik, CommonPaint nevezetű részbe. Ennek az új metódusnak két paramétere van, a vászon, és a nagyítási faktor, a végeredménye pedig az átadott vásznon megjelenő megfelelő méretezésű alaklista:

```
procedure TShapesForm.CommonPaint (
  Canvas1: TCanvas; Scale: Integer);
var
  I: Integer;
  CurShape: ShapeData;
begin
  for I := 0 to ShapesList.Count - 1 do
  begin
    CurShape := ShapesList.Items;
    with CurShape do
    begin
      Canvas1.Pen.Color := PenColor;
      Canvas1.Pen.Width := PenSize;
      Canvas1.Brush.Color := BrushColor;
      if Circle then
        Canvas1.Ellipse (
          (X-Size) * Scale, (Y-Size) * Scale,
          (X+Size) * Scale, (Y+Size) * Scale)
      else
        Canvas1.Rectangle (
          (X-Size) * Scale, (Y-Size) * Scale,
          (X+Size) * Scale, (Y+Size) * Scale);
    end;
  end;
end;
```

Ennek elkészítése után a FormPaint és a Print1Click metódusok megírása már egyszerű. A képernyőre rajzoláshoz elég, ha meghívjuk a CommonPaint eljárást, nagyítási faktor nélkül:

```
procedure TShapesForm.FormPaint(Sender: TObject);
begin
  CommonPaint (Canvas, 1);
end;
```

Hogy a form képét a nyomtatón megjelenítsük, a végeredményt reprodukálnunk kell a nyomtató vásznán, a megfelelő átméretezéssel. Ahelyett, hogy a felhasználóra hagytuk volna a nyomtatási méret kiválasztását, automatikusan fogunk számolni. Az ötlet az, hogy az alakzatokat a lehető legnagyobb méretben jelenítsük meg, azáltal, hogy a form kliens területét akkorára nagyítjuk, hogy az elfedje az egész oldalt. Ennek megvalósítása talán egyszerűbb a leírásánál:

```
procedure TShapesForm.Print1Click(Sender: TObject);
var
  Scale, Scale1: Integer;
begin
  Scale := Printer.PageWidth div ClientWidth;
  Scale1 := Printer.PageHeight div ClientHeight;
  if Scale1 < Scale then
    Scale := Scale1;
  Printer.BeginDoc;
  try
    CommonPaint (Printer.Canvas, Scale);
    Printer.EndDoc;
  except
    Printer.Abort;
    raise;
  end;
end;
```

Természetesen, nem szabad elfelejtenünk, hogy a nyomtatás megkezdéséhez szükség van a BeginDoc parancsra, a kész kép elküldéséhez pedig az EndDocra, és ezek közé kell kerülnie a CommonPaint metódusnak. Ha kivétel keletkezik, akkor meghívjuk az Abort függvényt, hogy mindenképpen leállítsuk a nyomtatást.

Szöveg nyomtatása

Az előző két példában a globális nyomtató objektum vásznát használtuk grafika nyomtatására. A fejezet elején láttunk egy példát, amivel szöveget nyomtathattunk, de annak a kódja messze volt a tökéletestől, pedig szöveg nyomtatása nagyon fontos dolog, érdemel némi figyelmet.

Delphi programból történő szövegnyomtatásra egyféle megoldás az, amit a PrnForm példában használtunk. Ha a szöveget egy formon, vagy általánosabban egy képen elkészítjük, akkor az így elkészült ábrát később kinyomtathatjuk a képernyőre. Ez néha hasznos lehet, például, ha egy adatbeviteli formunk van, de általában nem megfelelő, mert csak korlátozott mennyiségű szöveget nyomtathatunk, és a nyomtatás minősége is gyenge.

Másik megközelítés lehet, hogy a szöveget a vászon TextOut metódusával, vagy más szövegekkel kapcsolatos rajzoló metódusokkal készítjük el. Ez

precíz irányítást enged a szöveg elhelyezkedése felett, és nagy felbontású végeredményt kaphatunk. A hátránya, hogy sokat kell vele dolgoznunk, mert nekünk kell meghatározni az oldal minden sorának hosszát és elhelyezkedését, a betűk magasságát, és sok más dolgot is, de a lehetőségekért meg kell szenvedni.

Van azonban két könnyítési lehetőség. Az első, hogy a szöveget egy RichEdit vezérlőbe írjuk, és meghívjuk ennek a Print metódusát. Erre láthattunk példát a 9. fejezet RichNot2 programjában. A másik megközelítés az, ha a nyomtatót szövegállományként használjuk.

Delphiben egy állományt rendelhetünk a nyomtatóhoz, ezután ebbe az állományba írhatunk – vagyis a nyomtatóra küldhetjük a szöveget – a szabványos Write és WriteLn eljárásokkal. Ez sokkal egyszerűbb, mint az összes eddigi metódus, mert a rendszer automatikusan eldönti a sorok magasságát, ezzel megkímélve minket rengeteg munkától. A nyomtatási minőség jó, bár nincs sok beleszólásunk a végeredménybe. A hosszú sorok automatikusan tördelésre kerülnek, ami hasznos, de nagyon vad dolgokat is eredményezhet.

Ennek a példának az elkészítéséhez felújítottuk a Notes példát a nyolcadik fejezetből. Ez nagyon egyszerű példa volt, ezért adtunk hozzá állománykezelő szolgáltatásokat, betűkészlet és szín párbeszédablakokat, és sok minden mást. Egy ilyen példában, ami egyetlen betűtípus használ az egész szövegre, és nincsenek különleges, oldalkezelő tulajdonságai, mint amilyen az oldalszámozás, fejléc, vagy lábléc, a Writeln eljárás használata egy, a nyomtatóhoz rendelt állományra, teljesen indokolt.

Ennek a megközelítésnek a kulcsa az AssignPrn eljárás, ami az állományt a nyomtatóhoz rendeli. A nyomtatási eljárás elindítása után a Write és Writeln eljárásokkal elkezdhetjük nyomtatni a szöveget a Rewrite segítségével. A PrnNotes példában a Memo komponens sorait nyomtatjuk ki egy ciklussal, ami az első sortól halad az utolsóig:

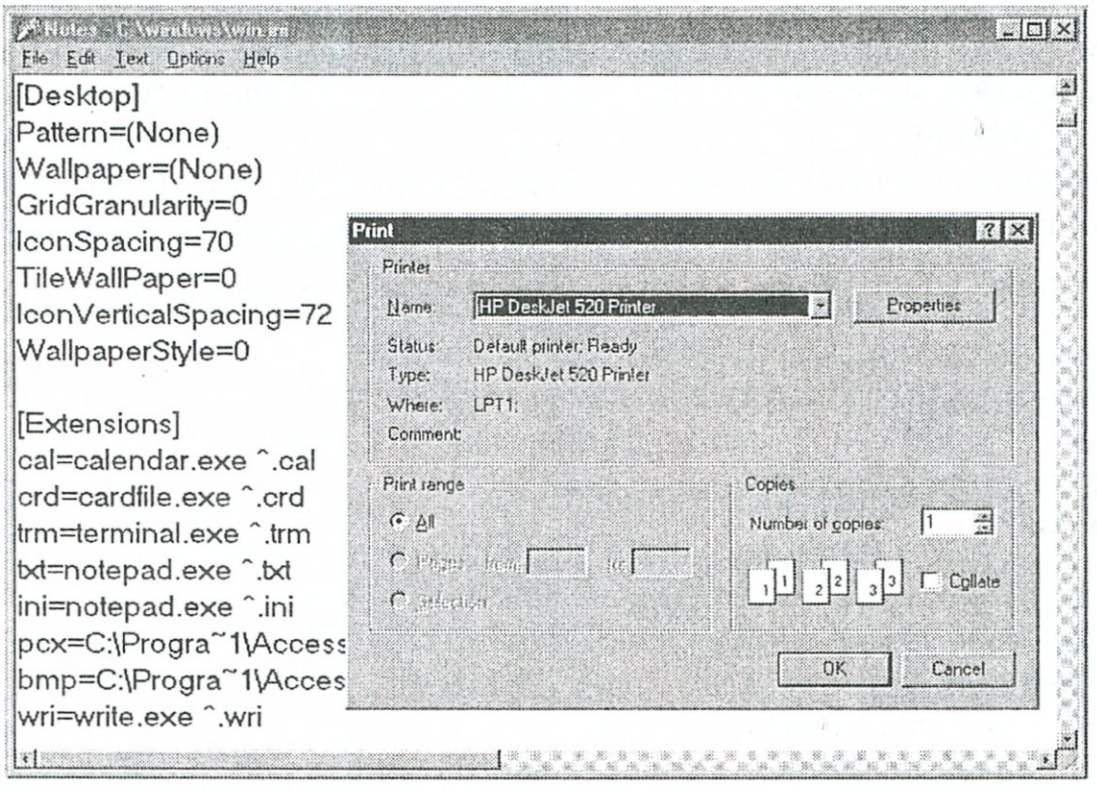
```
procedure TNotesForm.Print1Click(Sender: TObject);
var
  PrintFile: TextFile;
  I: Integer;
begin
  if PrintDialog1.Execute then
  begin
    {assign the printer to a file}
    AssignPrn (PrintFile);
    Rewrite (PrintFile);
    try
      {set the font}
      Printer.Canvas.Font := Memo1.Font;
      {copy all the text of the memo to the
      printer file}
```

```
          for I := 0 to Memo1.Lines.Count - 1 do
            Writeln (PrintFile, Memo1.Lines );
        finally
          CloseFile (PrintFile);
        end;
      end;
    end;
```

Figyelem!

Ha egy komponens, vagy form által használt betűtípus meg kell jelenítenünk a nyomtatón, akkor gondoljunk arra, hogy nem minden képernyő betűtípus érhető el nyomtatásban is. Erre példa az általánosan használt MS Sans Serif betűkészlet, amit nem lehet rendesen kinyomtatni, emiatt ajánlott nyomtatásokkal kapcsolatos dolgoknál csak TrueType betűtípusokat használni, mert ezek ugyanúgy néznek ki képernyőn és nyomtatón is.

Könnyedén testre szabhattuk volna ezt a nyomtatási függvényt egy párbeszédablakkal, amiben beállíthatjuk a kinyomtatandó sortartományt, de inkább a 28.6. ábrán látható szabványos Print párbeszédablakot használtuk.



28.6. ábra *A PrnNotes példa szabványos nyomtatás párbeszédablaka*

Az oldaltartományra vonatkozó kiválasztás sokkal összetettebb dolog. Először ki kell számolnunk az egy oldalra kerülő sorok számát a nyomtató PageHeight (oldalmagasság) és a nyomtató vásznának betűtípus tulajdonságából. Ezután meg kell néznünk, hogy mely sorok lógnak túl a következő sorba (vagy következő sorokba), és ennek megfelelően kell módosítanunk a megadott oldalra kerülő sorok számát.

Amikor új oldalt kezdünk, akkor nem hívhatjuk meg a nyomtató objektum EndPage metódusát. Ha így tennénk, akkor a nyomtató ugyan kidobná a lapot, de a következő sor ugyanúgy a lap közepére kerülne, mintha semmi sem történt volna. Nyomtatóhoz kapcsolt állományok esetén, ha új oldalt akarunk, egyszerűen zárjuk le a kimeneti állományt, és nyissuk meg újra. Az állomány lezárása ugyanis lezárja az aktuális oldalt, az új oldal pedig az első sortól fog kezdődni (az oldalszám és a fejléc után).

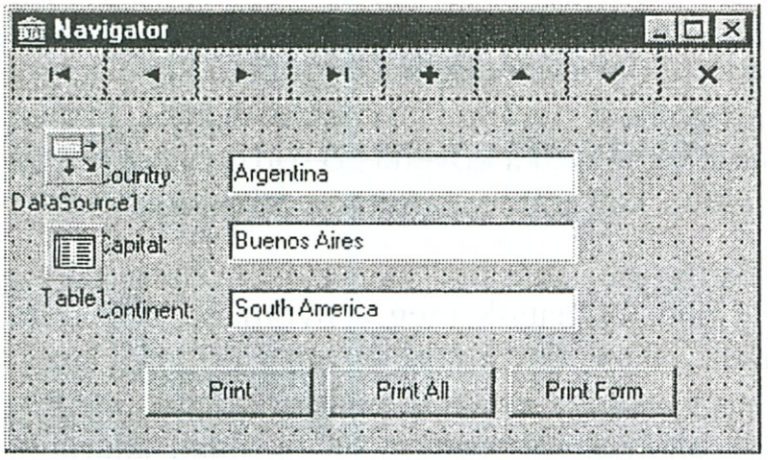
Adatbázisrekordok és -táblák nyomtatása

A Delphi nyomtatási képességeinek felhasználásával könnyedén elkészíthetjük a programunk adatbázis formjainak nyomtatott képét. Hiszen, ha egy form egy adatbázis tábla rekordjait mutatja, akkor a form Print metódusával megjeleníthetjük az adatokat. Természetesen, ha nyomtatni akarjuk az adatokat, akkor már a formot is célszerű úgy elkészíteni, hogy a kép jól nézzen ki.

Másik megoldáskén kinyomtathatjuk a form teljes kliens területét bitképként, ehhez azonban készítenünk kell egy nyomtatási függvényt, ami átmenetileg eltávolítja az eszköztárat és a státuszsort, kinyomtatja a form egész felületét, és ezután visszaállítja az eszköztárat, és a státuszsort. Ennél jobb megoldás, ha egy nyomtatási kép formot nyitunk meg, amire rámentjük a form képét, majd ennek csak egy részét nyomtatjuk ki, kihagyva az eszköztárat, a státuszsort, és a többi extra elemet.

Nem ez az egyetlen megoldás a rekordok nyomtatására – vagy akár teljes táblák nyomtatására, feltéve, ha elférnek egy képernyőn – adatbázis formok esetén. Az előző részben, a szöveg nyomtatására használt valamennyi metódus használható. Az állományba nyomtatással például az oldal minden sorába leírhatjuk a mező nevét, és az értékét, így kinyomtatva, egy, vagy több rekordot.

Az első, adatbázis nyomtatással kapcsolatos példánk a PrintNav, ami a tizenhatodik fejezet Navig1 példájának bővítése, és hasonlít még a huszonkettedik fejezet DbToWord, és a huszonnegyedik fejezet DbToHtml példájára is. Ehhez a két utolsó változathoz hasonlóan a PrintNav példának is van néhány gombja, ahogy az a 28.7. ábrán látszik.



28.7. ábra *A PrintNav példa formja tervezés alatt*

A három gomb különböző, nyomtatással kapcsolatos műveleteket végez. Az első egyszerűen kinyomtatja az aktuális rekord két látható mezőjét a két címke, és a két szerkesztőmező szövegének felhasználásával:

```
procedure TNavigator.PrintButtonClick(Sender: TObject);
var
  PrintFile: TextFile;
begin
  {assign the printer to a file}
  AssignPrn (PrintFile);
  Rewrite (PrintFile);
  try
    {set the font of the printer, and output
    each element}
    Printer.Canvas.Font := Font;
    Writeln(PrintFile,Label1.Caption,' ',DBEdit1.Text);
    Writeln(PrintFile,Label2.Caption,' ',DBEdit2.Text);
    Writeln(PrintFile,Label3.Caption,' ',DBEdit3.Text);
  finally
    {close the printing process}
    CloseFile (PrintFile);
  end;
end;
```

Ez a kód követi a PrintFile állomány hozzárendelését a nyomtató kimenethez. Ami új a metódusban, az az, hogy az adatbázis hozzáférések, és a kiíratő műveletek egy `try-finally` blokkon belül helyezkednek el, hogy nehogy kimaradjon a nyomtatási munkát lezáró rész valamilyen hiba folytán. A második gomb kódja, a Print All (mindent nyomtat) sokkal bonyolultabb, ez ugyanis az összes adatbázisrekord látható mezőit kiírja, nem csak az aktuálisét, vagyis a programnak végig kell mennie az egész táblán, mint a DbToWord, vagy a DbToHtml példában. Ahogy az lentebb látszik, az elején beállítunk egy

könyvjelzőt, kikapcsoljuk a kontrolokat, és végigmegyünk a táblán egy elöltesztelő ciklussal. A program minden tételre kiírja a címkét a mező szövegével. A program a mezőket a tábla FieldByName (mező név alapján) metódusával szedi ki, és a TField osztály AsString metódusával alakítja szöveggé. Íme az eljárás teljes forráskódja, amiben három egymásba ágyazott `try-finally` blokk van, hogy mindenféle lehetséges kivételre felkészüljünk:

```
procedure TNavigator.PrintAllButtonClick(
    Sender:TObject);
var
  Bookmark: TBookmark;
  PrintFile: TextFile;
begin
  {assign the printer to a file}
  AssignPrn (PrintFile);
  Rewrite (PrintFile);
  try
    {set the font of the form}
    Printer.Canvas.Font := Font;
    {store the current position,
    creating a new bookmark}
    Bookmark := Table1.GetBookmark;
    try
      Table1.DisableControls;
      try
        Table1.First;
        while not Table1.EOF do
        begin
          {output the three fields and a blank line}
          Writeln (PrintFile, 'Country: ',
            Table1.FieldByName ('Name').AsString);
          Writeln (PrintFile, 'Capital: ',
            Table1.FieldByName ('Capital').AsString);
          Writeln (PrintFile, 'Continent: ',
            Table1.FieldByName ('Continent').AsString);
          Writeln (PrintFile);
          Table1.Next;
        end;
      finally
        Table1.EnableControls;
      end;
    finally
      Table1.GotoBookmark (Bookmark);
      Table1.FreeBookmark (Bookmark);
    end;
  finally
    System.CloseFile (PrintFile);
  end;
end;
```

A Print Form (form nyomtatása) gombnak van a legegyszerűbb kódja, ehhez a PrintFormButtonClick metódus egyszerűen meghívja a form Print metódusát. Ez a gomb tulajdonképpen csak azért került a programba, hogy kipróbálhassuk az adatbázis formok adatainak nyomtatására szolgáló három módot. Ezen metódusok bármelyikét kibővíthetjük fejlécekkel, megjegyzésekkel, más mezők kinyomtatásával, és így tovább. Ez a példa csak arra szolgál, hogy megmutassa, milyen kódot lehet ilyenkor használni.

Ennek az ötletnek a nyomán összetett, és látványos nyomtatványokat, jelentéseket készíthetünk. Másik megoldásként választhatjuk a Delphi beépített jelentésgeneráló eszközeit, a QuickReport komponenst, vagy a ReportSmith-t.

A QuickReport komponensek

A Delphi Professional változata tartalmazza a QuickReportot, jelentéskészítő komponensek gyűjteményét, amik szorosan a Delphihez kapcsolódnak, és amiket a Borland a QSD AS-től, egy norvég cégtől vett át. Hasonló Delphi komponenseket más külső cégek is készítettek, de mi most ezekkel foglalkozunk, mert ezek a legszélesebb körben elérhetőek a Delphi fejlesztők számára.

Megjegyzés

A QuickReport mellett más jelentéskészítő komponens gyűjtemények is léteznek, mint amilyen a ReportPrinter, a Piparti, az ACE Reporter, vagy sok más hasonló, amik közül néhány példa még kísérő CD-nken is rajta van.

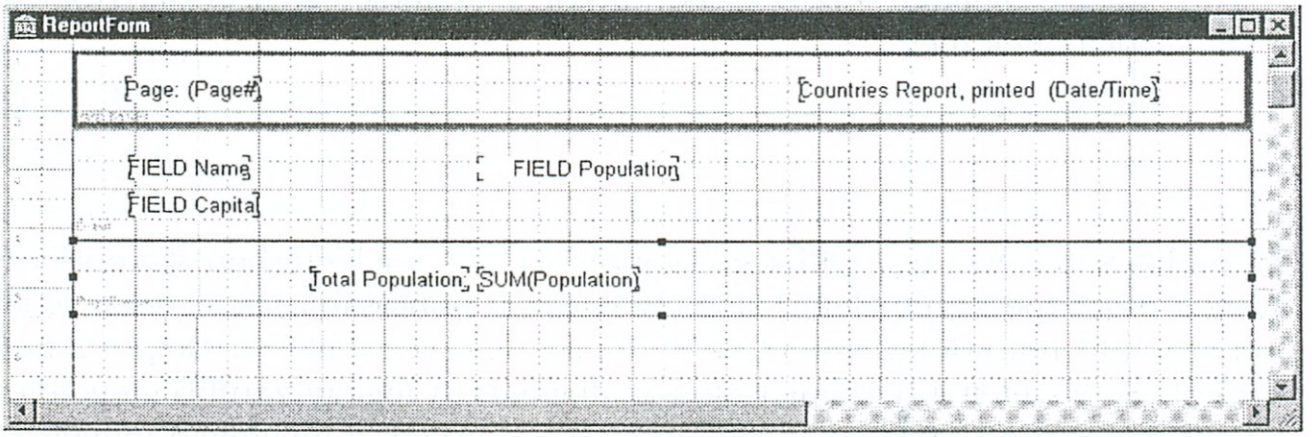
A QuickReport formot használ a jelentés tervezéséhez, és sok szempontból hasonló a normális formok elkészítéséhez. Ezeket a formokat azonban csak jelentések készítésére használhatjuk, futásidőben soha nem jelennek meg a képernyőn. Hogy kinyomtassuk, vagy megjelenítsük a jelentést, meghívhatjuk a minden jelentés formra letett QuickReport (ha egy formra QuickReport komponenst teszünk, akkor az a form jelentésformmá válik) komponens Print vagy Preview metódusait.

A QuickReport használatával a jelentésünk sávokból épül fel. A sávok vízszintes információs területek. Sávokkal adatokat jeleníthetünk meg, minden oldalra fej és láblécet tehetünk, betehetünk összegeket, vagy más különleges információkat. Jelentéskészítéséhez tegyünk a QuickReport komponenst egy másodlagos formra – ne a fő formra – adjunk hozzá egy vagy több sávot, ezután tegyünk a sávokra néhány adat érzékeny QuickReport komponenst; ezek

olyan komponensek, amik a szokásos módon kapcsolódnak a Delphi adatforrásokhoz. Az adatok érkezhetnek egy vagy több táblából vagy lekérdezésekből, épp úgy, mint a szabványos adathozzáférési komponenseknél.

Egy gyors példa

Hogy bemutassuk a QuickReport komponenst, kibővítettük az előző példa (PrintNav) főformját, egy teljes jelentés hozzáadásával. Az új példát QrNavnak neveztük el. Amellett, hogy a új gombot tettünk az eddigi három nyomtatásra szolgáló mellé, egy új, ReportForm nevű, formot is adtunk a projekthez. Az új formon egy QuickReport és három QRBand komponens van, ahogy az a 28.8. ábrán is látható.



28.8. ábra *A QrNav példa elkészítéséhez használt form tervezés alatt. Ezt a formot a rajta lévő jelentés komponensek használják, és soha nem jelenik meg futás közben*

A QRBand egyik kulcs tulajdonsága a BandType (sávtípus), ami a sáv szerepét határozza meg a jelentésben. Ebben a példában az első sáv rbPageHeader (oldalfejléc), a második rbDetail (részletek), a harmadik pedig rbPageFooter (oldal lábléc) típusú. Más használható sávtípusok például a rbTitle (cím), ami az első oldal fejléce elé kerül, az rbSummary (összegzés), ami a jelentés végére kerül, az rbGroupHeader (Csoportfej) és az rbGroupFooter (Csoportláb), amik a különleges, QRGroup (csoport) komponens által meghatározott komponensekre vonatkoznak, a rbColumnHeader (oszlopfej), ami többoszlopos jelentéseknél jelentkezik, és még néhány másik.

A példában két QRSysData komponenst tettünk az első sávba (az oldalfejre). A QRSysData komponensek jelenítik meg az oldalszámot, az időt, a dátumot, és a Text tulajdonságában lévő leírást. Még sok más rendszerinformációt jeleníthetünk meg ezekkel a komponensekkel, ahogy azt a Data tulajdonságuk jelzi. Szintén ezzel állítottunk be keretet a bennfoglaló sávnak, a Frame (keret) tulajdonsággal.

A második sáv tartalmazza az adatbázisból származó valódi adatokat. Részlet sávként, újra és újra megjelenik, minden egyes, az adatforrásból származó rekordnál, ezért minden jelentés komponensnek adatforrással kell rendelkeznie. Ebben az esetben a főform Table1 komponensét használtuk (a **File** menü **Use Unit** parancsa után). Ez az adatkészlet csatlakozik a második sáv három QRDBText komponenséhez is.

Tipp

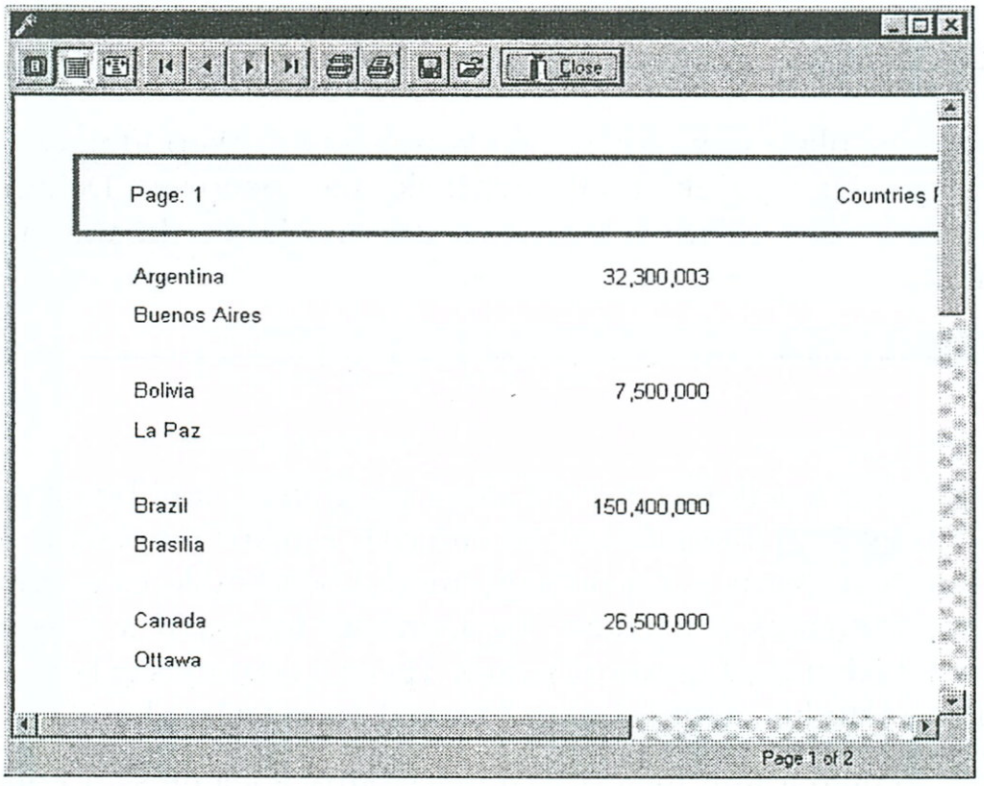
A szabványos Delphi adatkomponensekkel megosztott DataField tulajdonság mellett a jelentés komponensnek még minimális formázási lehetőségei vannak. Ha például a `Mask` tulajdonságot erre az értékre állítjuk: '###,###,###', akkor a számok ezres elválasztókkal jelennek meg.

Az utolsó sávon van egy QRExpr komponens, ami a jelentésben szereplő országok össznépességét jeleníti meg (ez egészen pontosan csak az adott oldalon szereplő rekordokra vonatkozik). Ez a komponens eltér a Delphi 2-ben lévőtől, és összetettebb számításokra is alkalmas. A legegyszerűbb megoldás, ha az Expression (kifejezés) tulajdonságában jelezzük a megfelelő műveletet (mint a sum, vagyis összegzés, min, max, average, azaz átlag, count, vagyis számosság), és mezőt (mint a Population, vagyis népesség). Az Expression tulajdonságnak van egy szerkesztője, amivel kifejezéseket szerkeszthetünk, ahelyett, hogy begépelnénk őket. Ne felejtsük azonban beállítani a Master tulajdonságot a QuickRep1 komponensre, mert ez az egyetlen módja, hogy a számított értéket a megfelelő adatkészlethez csatoljuk. A jelentés egy formon alapul, ezért nézzük a form szöveges leírását:

```
object ReportForm: TReportForm
  object QuickRep1: TQuickRep
    DataSet = Navigator.Table1
    ReportTitle = 'Countries Report'
    object QRBand1: TQRBand
      BandType = rbPageHeader
      object QRSysData1: TQRSysData
        Data = qrsPageNumber
        Text = 'Page: '
      end
      object QRSysData2: TQRSysData...
    end
    object QRBand2: TQRBand
      BandType = rbDetail
```

```
        object QRDBText1: TQRDBText...
        object QRDBText27: TQRDBText...
        object QRDBText3: TQRDBText
          DataSet = Navigator.Table1
          DataField = 'Population'
          Mask = '###,###,###'
        end
      end
      object QRBand3: TQRBand
        BandType = rbPageFooter
        object QRExpr1: TQRExpr
          Expression = 'SUM(Population)'
          Mask = '###,###,###'
        end
        object QRLabel1: TQRLabel...
      end
    end
  end
```

Miután megterveztük a jelentést, le is tesztelhetjük azzal, hogy kettőt kattintunk a jelentés komponensen. Ez a form előzetes képét mutatja meg, amit felhasználhatunk nyomtatásra a program lefordítása nélkül is. Ugyanezt a nyomtatási képet nyerhetjük futás időben, ami a 28.9. ábrán látszik is, ha a `ReportForm.QuickReport1.Preview` beírjuk a gomb kódjába.



28.9. ábra *A QrNav példa QuickReport komponensen alapuló nyomtatási kép formja*

ReportSmith jelentések készítése

A VCL alapú jelentés komponensek mellett használhatunk külső jelentéskészítőket, mint a ReportSmith, vagy a Cristal Report. Ebben a részben egy ReportSmith-es példát mutatunk be. Azért választottuk ezt, mert korábbi Borland termékként még mindig része a Delphi Client/Server változatának, valamint a Delphi rendelkezik illesztő felülettel ehhez, a Report komponens képében.

Maga a komponens nem túl összetett. Tulajdonságai közé tartozik a nyomtatandó jelentésállomány neve, a könyvtára, és néhány más információ. A Report komponens Run metódusának meghívásával nézhetjük meg (a ReportSmith futásidejű moduljával), és nyomtathatjuk ki a jelentésünket. A ReportSmith önálló változata azonban nagyon sokoldalú jelentéskészítő eszköz, amit magában, tehát a Delphi nélkül, is használhatunk. A ReportSmithnek több szolgáltatása, és jobb tulajdonságai vannak, mint a QuickReport komponensnek, de rengeteg futtató DLL-re van szüksége, amit a programunkkal együtt kell telepítenünk, valamint sok memóriát igényel a megfelelő sebesség eléréséhez.

Ezzel szemben a QuickReport komponensnek nem kellenek extra DLL-ek (hacsak nem akarjuk a megfelelő csomagot önálló DLL-ként használni), és nagyon gyors, de korlátozottak a képességei. A QuickReport jobban összeépült a Delphivel, az adatbázisokat a Delphi Table és Query komponenseivel éri el, a ReportSmith pedig saját, független megközelítést használ.

Megjegyzés

A fent összehasonlítás nem korlátozódik csak a QuickReportra, és a ReportSmithre. Ugyanezek a különbségek más beépített Delphi komponensek, és külső jelentéskészítő modulok között is megvannak.

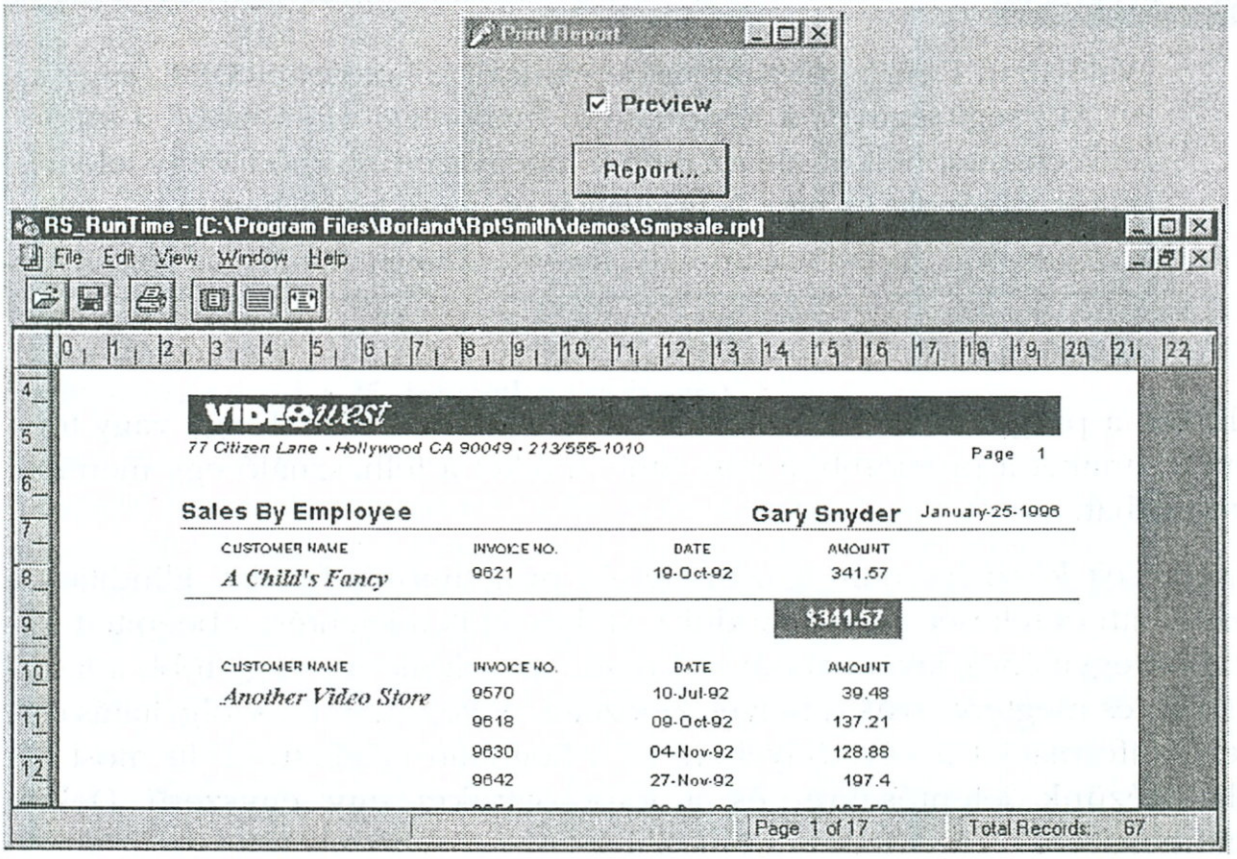
A ReportSmith vizsgálatára írtunk egy RsPrint nevű programot, ami betölt és lefuttat egy jelentést (egy RPT kiterjesztésű állományt). Amikor meghívjuk a Report1 objektum Run metódusát, a ReportSmith modul betöltődik, és ezután láthatjuk, és ki is nyomtathatjuk a jelentést. Ez azért van így, mert a Report komponens Preivew tulajdonságát igazra állítottuk. Ha ez a tulajdonság hamis, akkor a jelentés a Run metódus meghívásakor azonnal nyomtatásra kerül.

Másik megoldásként, miután a Run metódussal betöltöttük a jelentést, a Print metódust felhasználhatjuk arra, hogy Dinamikus Adat Csere (DDE) üzenetet küldjünk a ReportSmith modulnak, amivel megkérjük, hogy nyomtasson. Ne

hívjuk meg a Printet a jelentés betöltése előtt, mert nem lesz semmi hatása. A PrintRpt példa formján négy komponens van: egy „állomány megnyitás" párbeszédablak, amiben a felhasználó kiválaszthat egy jelentést, egy Report komponens, egy jelölőnégyzet, amiben a nyomtatási kép módot jelölhetjük, és egy gomb, ami elindítja a műveletet. Amikor a Report gombra kattintunk, a program megnyitja a párbeszédablakot, amiből kiválaszthatunk egy állományt, és elindítja a modult:

```
procedure TForm1.ReportButtonClick(Sender: TObject);
begin
  Report1.Preview := PreviewCheckBox.Checked;
  if OpenDialog1.Execute then
  begin
    Report1.Reportname := OpenDialog1.FileName;
    Report1.Run;
  end;
end;
```

Mivel beállítottuk az AutoUnload tuladjonságot, a ReportSmith futásidejű modulja automatikusan kitörlődik a memóriából amikor a jelentés befejeződik, másként a form OnClose eseményében el kellene helyezni egy különleges kifejezést, ami a Report komponens CloseApplication metódusát meghívja.



28.10. ábra *A jelentés előzetese a ReportSmith futás idejű ablakában jelenik meg. Felette az RsPrint alkalmazás látható*

Ahogy azt korábban említettük, a fenti kód hatása a Print Preview jelölőnégyzet állapotától függ. Ha a Nyomtatási kép tulajdonság hamis, akkor a ReportSmith modul automatikusan nyomtatni kezd, egyébként pedig a ReportSmith futásidejű ablak jelenik meg, ami a 28.10. ábrán is látható. A ReportSmith ablak azon a pozíción jelenik meg, ahol a legutóbbi használatkor volt, és az előzetes képet mutatja. Itt különböző változásokat eszközölhetünk, kiválaszthatjuk a nyomtatandó oldalakat, és más lehetőségeket is.

A Delphi és a ReportSmith modul közti kapcsolat nagyon laza. A ReportSmith önálló jelentéskészítő alkalmazás, amit a Delphi is tud vezérelni, de önállóan is használható. Azt viszont nem tehetjük meg, hogy a jelentés nyomtatási képét egy Delphi ablakban jelenítsük meg, a következő példában azonban pont ezt fogjuk megcsinálni.

Saját jelentés készítése

Az RsPrint program segítségével megnézhetünk, és kinyomtathatunk már létező jelentéseket, de nem készíthetünk újakat, vagyis, ha ezt a megközelítést használjuk, akkor a programunk felhasználói nem fognak tudni saját jelentéseket készíteni, hacsak nincs egy példányuk a ReportSmith-ből.

Megjegyzés

Valójában Delphiből is készíthetnénk új jelentéseket DDE, vagy az új API segítségével, a ReportSmith futás idejű moduljával, a különálló csomag nélkül, de ez nem a legegyszerűbb módja egy jelentés elkészítésének, és nem is tárgyaljuk ebben a könyvben.

Általában, a programunk fejlesztése során el kell készítenünk egy vagy több jelentést, amiket a programhoz csatolunk, amiket a felhasználó egy menüből kinyomtathat.

Új jelentések készítése nem igazán Delphi programozói feladat. Elindítjuk a ReportSmith (a teljeset, nem a modult), előkészítjük a jelentést a beépített formátumok egyikének kiválasztásával, hozzákapcsolunk egy vagy több adatbázis táblát, és megtervezzük a helyes kinézetet. A ReportSmith szolgáltatásairól bővebb információt a kézikönyvben, és a Súgóban találhatunk. Itt most két példát nézünk jelentésekre, és a kapcsolatukra egy egyszerű Delphi programmal.

Az első ReportSmith példánk a szintén a tizenhatodik fejezetből már ismert HandGrid egy újabb változata lesz. Ez, a RsGrid nevű új változat ugyanazt a

táblázatot tartalmazza, mint az előző, de van egy menüje, amiből több féle jelentést nyomtathatunk. A program felépítése egyszerű. Van egy, a COUNTRY.DB táblához kapcsolódó DBGrid. A programban van egy egyszerű nyomtatási parancs, ami egy egyszerű párbeszédablakot használ, hogy leellenőrizze a felhasználó szándékát:

```
procedure TForm1.Print1Click(Sender: TObject);
begin
  if PrintDialog1.Execute then
    Print;
end;
```

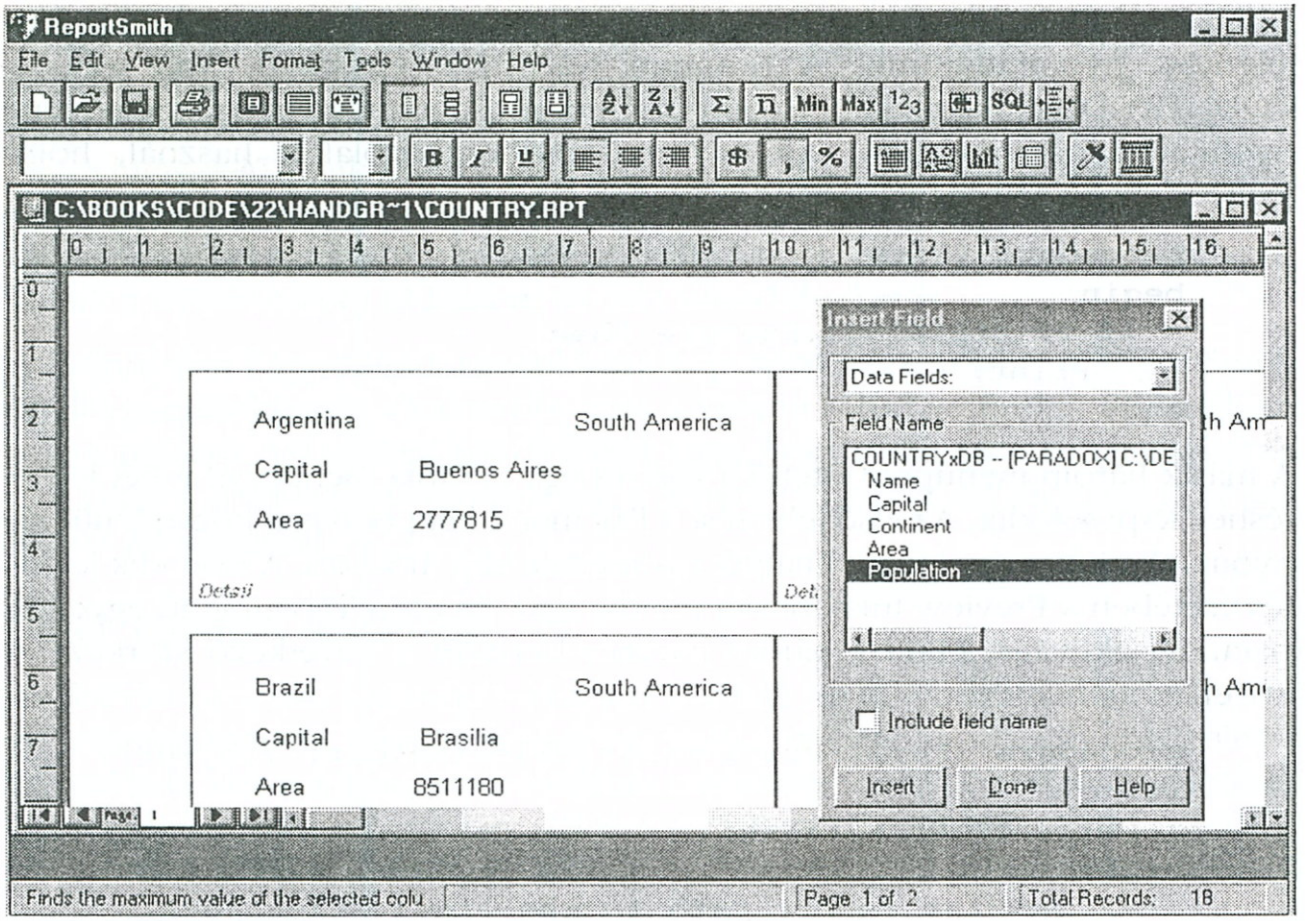
A másik három menüpont (kettő a Fájl, és egy a Tábla menüben) a két jelentéshez kapcsolódik. Az első jelentést a Fájl menü két pontjával közvetlenül kinyomtathatjuk, vagy megnyithatjuk nyomtatási kép módban. A második jelentés esetében a Preview tulajdonság mindig igaz, és ezt a Tábla menü egyetlen parancsa aktiválja. Mind a három parancs kezelője a következő szerkezetet követi:

```
procedure TForm1.PrintReport1Click(Sender: TObject);
var
  CurrentPath: String;
begin
  // set the report path to the application path
  CurrentPath := ExtractFilePath (ParamStr(0));
  Delete (CurrentPath, 1, 1);
  Report1.ReportDir := CurrentPath;
  Report1.Preview := False;
  Report1.Run;
end;
```

A jelentés kinyomtatása előtt be kell állítanunk a helyes könyvtárat. Mivel nem akarunk rögzített könyvtárneveket, ezért célszerűbb, ha ezt az alkalmazás teljes útvonalából állapítjuk meg, amit első paraméterként kapunk a program indításakor (erről a 25. fejezetben volt szó).

Ami a második két esetben megváltozik, az a Preview tulajdonság értéke, és a használt Report komponenst. A form, és a program kódja egyszerű, hiszen a jelentések elkészítése a ReportSmith-szel történt. Az első jelentésről a következő bekezdésben lesz szó, a másodikról utána, mert a második jelentés kihasználja a ReportSmith néhány komolyabb szolgáltatását.

Az első jelentés, amit a RsGrid példának készítettünk, a COUNTRY.RPT, nem tesz mást, mint megjeleníti az adatbázis tartalmát címke formátumban. Hogy ezt újra elkészítsük, indítsuk el a ReportSmith-t, készítsünk egy új jelentést, válasszuk ki a címkeformátumot (label style), válasszuk ki a megfelelő táblát – itt most nem érhetjük el a BDE álneveket. Ezután húzzunk néhány mezőt a jelentés oldalára, tegyünk rá egy címkét a megfelelő jelölőnégyzettel. A jelentés egyik munkaváltozata látható a 28.11. ábrán.



28.11. ábra *ReportSmithben jelentést készíteni mindössze néhány címke áthúzásából áll*

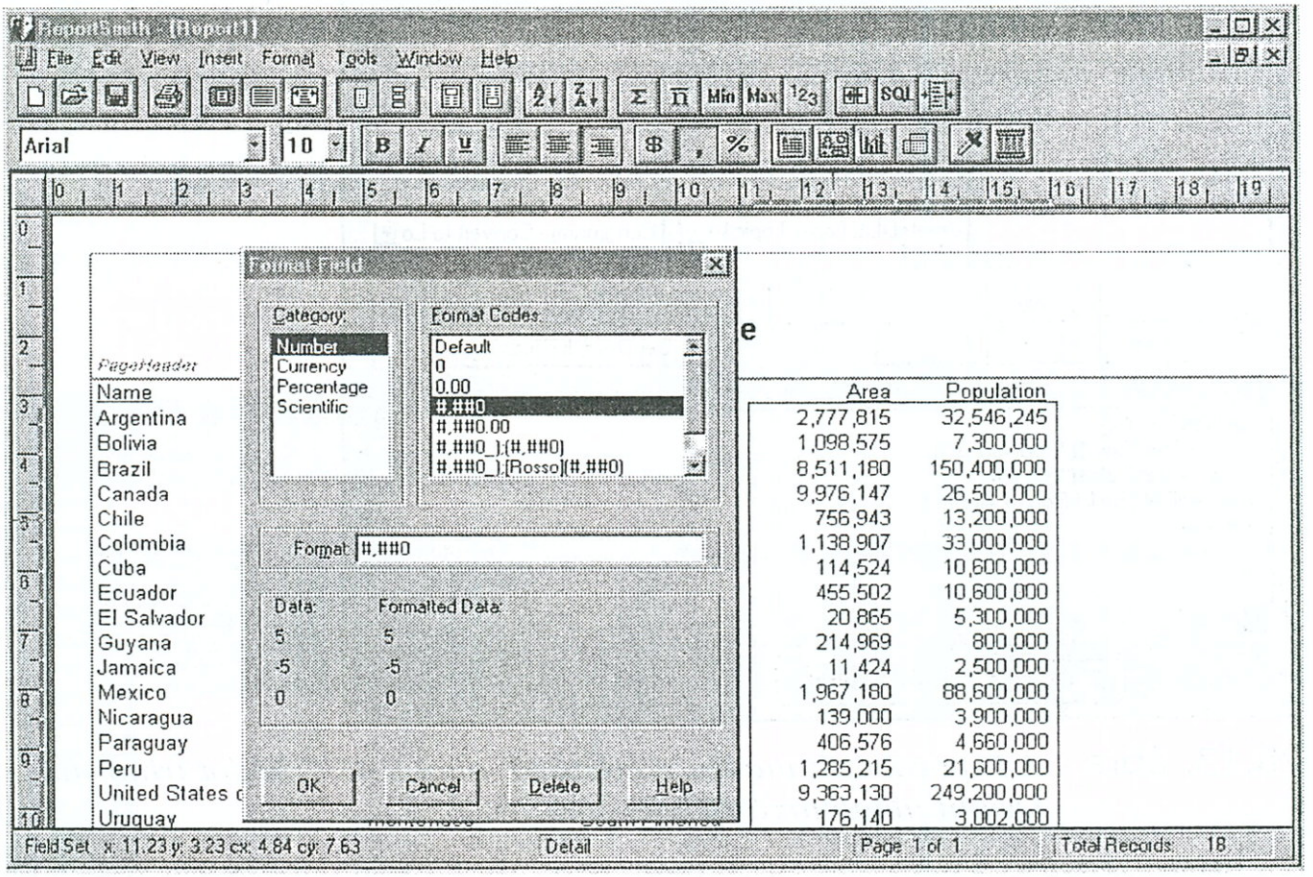
Amikor elkészültünk a jelentés kinézetével, mentsük el az állományt, és a nevét adjuk meg a Report1 komponens ReportName tulajdonságánál a HandGrid3 példában. Ezután futtathatjuk a programot, hogy megnézzük, vagy kinyomtassuk a jelentést, ez ilyen egyszerű.

Figyelem!

A jelentés állományban, és az RsGrid programban szerepel néhány beégetett könyvtárnév, ezek között azé a könyvtáré is, ami a jelentés elkészítéséhez szükséges táblákat tartalmazza. Ha a rendszerünk nem az alapértelmezett Delphi könyvtárakat tartalmazza, akkor gondjaink lehetnek a program futtatásával.

ReportSmith makró készítése

Az RsGrid példa második jelentése, a TABLE.RPT, valamivel összetettebb. Ez az oszlopos jelentés formátumot használja a címkeszerű helyett. Válasszuk ki ismét a COUNTRY.DB táblát, és az összes mezője oszlopokba rendeződik. Átméretezhetjük az oszlopokat, hogy elférjenek az oldalon, beállíthatjuk a helyes numerikus formázást, mint amilyenek az ezres elválasztók, ahogy azt a 28.12. ábrán láthatjuk.

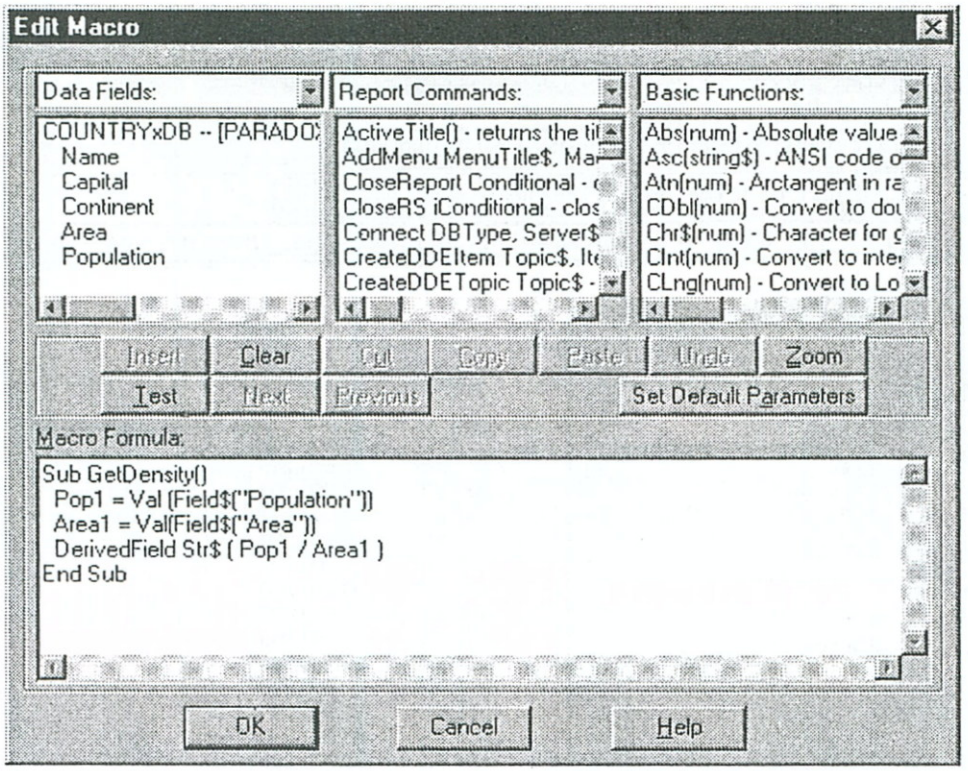


28.12. ábra *A jelentéskészítésének kezdő lépései az oszlopok átméretezése, és a megfelelő formázás kiválasztása*

A jelentéskészítését a megfelelő cím bevitelével fojtatjuk, lerendezzük kontinensek, és ország nevek szerint, és a jelentést kontinensenként csoportosítjuk. Ezután minden csoportnak adunk egy láblécet. Ebben a láblécben a kontinensenkénti ország számot, a kontinens összlakosságát, és teljes területét jelenítjük meg.

Végül átmehetünk a bonyolultabb részre: hozzá fogunk tenni egy számított mezőt, ami, a tizenhatodik, és tizenhetedik fejezetekben megszokott módon, a kontinens népsűrűségét tartalmazza. A probléma abból adódik, hogy a számított mező definíciójánál nem használhatunk SQL kifejezést, mert ezt a

ReportSmith nem támogatja Paradox táblák esetében. Az egyetlen lehetséges technika ReportSmith makró használata. A Tools (eszközök) menü Derived Fields (számított mezők) parancsával hozhatunk létre olyan új mezőt, amit egy ReportBasic makró határoz meg. Mi a Density (sűrűség) nevet használtuk, ezután elnevezhetjük a makrónkat, például GetDensity-nek (sűrűség meghatározás), és elkészíthetjük a kódját.



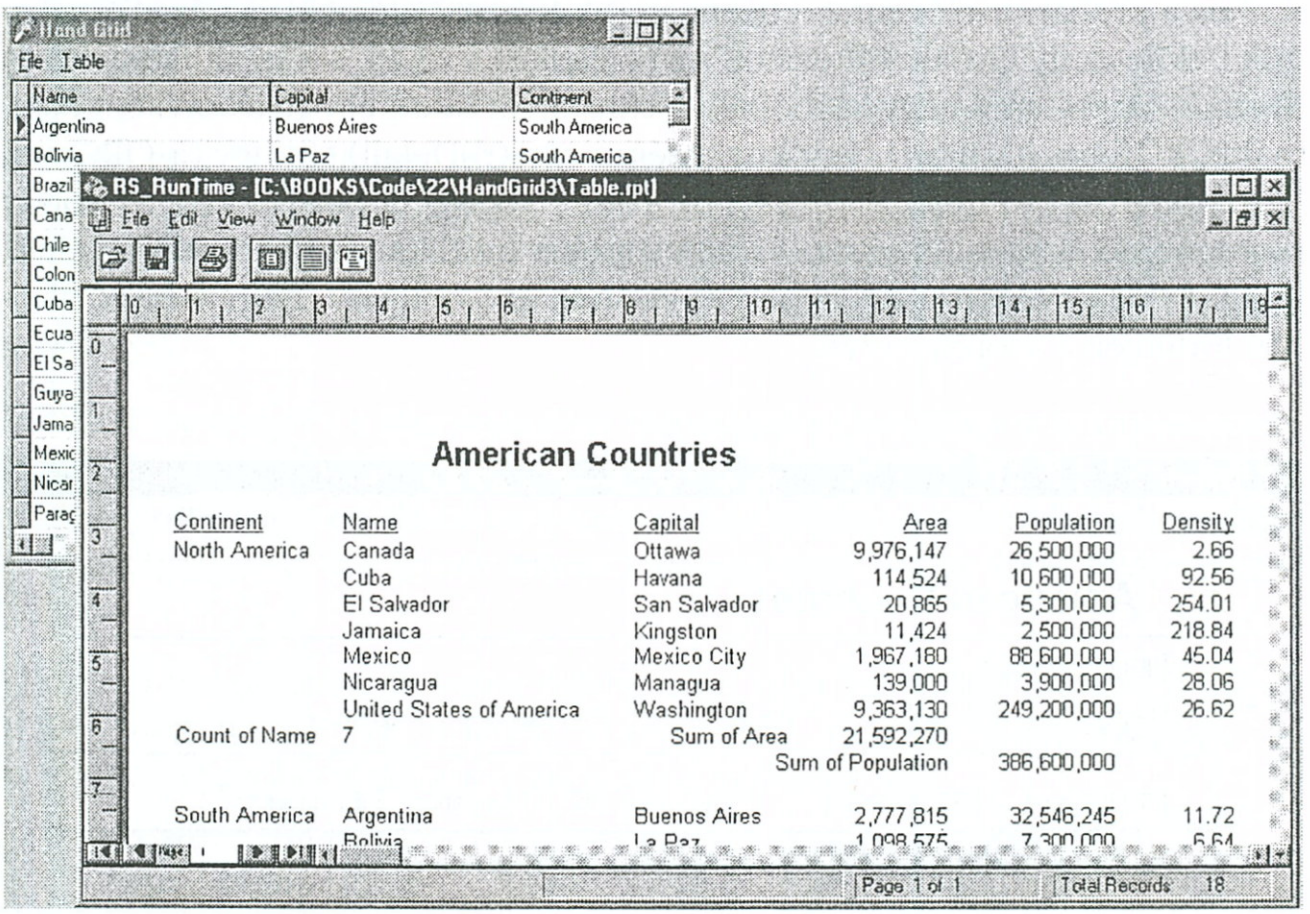
28.13. ábra *A ReportSmith makro szerkesztő, ahol BASIC kódot írhatunk Delphi alkalmazásokhoz*

Amikor a Choose a Macro (makró kiválasztása) párbeszédablakban a **New** (új) vagy **Edit** (szerkesztés) gombokra kattintunk, akkor a 28.13. ábrán látható Edit Macro párbeszédablak jelenik meg. A makró szövegét közvetlenül a makró szerkesztőbe is gépelhetjük, vagy a fenti három lista segítségével is kiválaszthatjuk a tábla mezőket, parancsokat és függvényeket. Mind a három lista fölött van egy kombinált lista, amiből a kapcsolódó listában lévő elemcsoportokat változtathatjuk. Például az utolsó lista négy csoportot jeleníthet meg: Basic Functions (alapvető függvények), Basic Statements (alapvető kifejezések), Dialog Box Functions (párbeszédablak függvények, vagy Branching & Looping (csoportosítás és ciklusok). Ezek a parancs listák hasznosak lehetnek makró íráskor, de ettől még mindig ismernünk kell a ReportSmith makró szerkesztő által használt parancsait (ez a ReportSmith Basic) legalább alapszinten. Általánosságban, ha a ReportSmith szolgáltatásait ki akarjuk használni, akkor némi BASIC alapismeret segíthet.

Az új mező makrója a következő:

```
Sub GetDensity()
   Pop1 = Val (Field$("Population"))
   Area1 = Val (Field$("Area"))
   DerivedField Str$ ( Pop1 / Area1 )
End Sub
```

Ez a kód két átmeneti változót definiál – a BASIC-ben nincs szükség deklarációra –, és a két mező értékét ezekbe a változókba helyezi. A Field$ függvény az aktuális rekord egyik mezőjének tartalmát adja vissza szöveges formában, a Val függvény pedig ezt alakítja szám formátumra. Az utolsó kifejezés kiszámítja a népsűrűséget, és visszaalakítja karakteressé a Str$ függvénnyel. Ez az érték kerül a DerivedField azonosítóba, ami automatikusan behelyettesítésre kerül a makró által számolt mezőbe. Ennek az eredmény látható a 28.14. ábrán, ami az RsGrid példa általam megjelenített végleges jelentést mutatja a ReportSmith futás közbeni ablakjában.



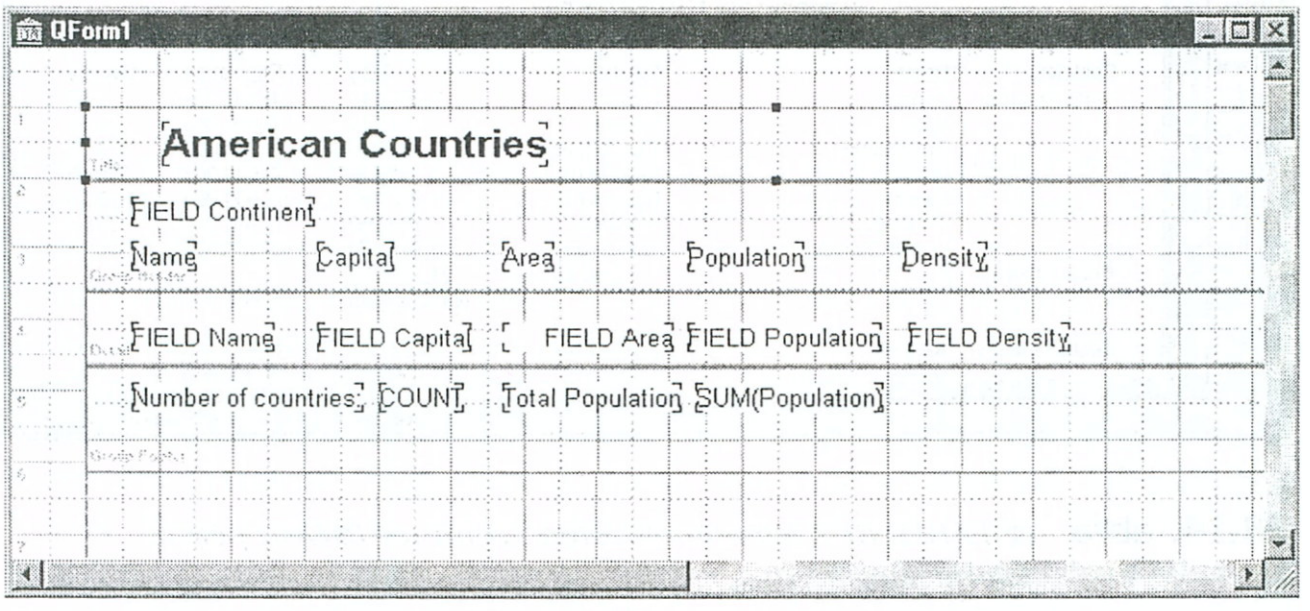
28.14. ábra *A TABLE.RPT jelentés előzetese, amit az RsGrid példa jelenít meg*

Gyorsabban a ReportSmithnél

Már láthattuk, hogy mennyi dologgal jár egy számított mezőket tartalmazó jelentés elkészítése a ReportSmithben. Amint az már említettük, ez a Delphi és a ReportSmith laza integrációjának tudható be. Van vajon jobb megoldás, mint amit az RsGrid példában használtunk? Igen, mégpedig kettő is. Az első, hogy a Delphivel készíttetünk egy olyan adatbázis táblát, ami csak a megjelenítendő adatokat tartalmazza. Ebben az esetben szükségünk lesz egy átmeneti táblára, ami a népsűrűség mezőket tartalmazza, ezután ebbe belemásolhatjuk az adatokat, még, az ezt használó jelentés elindítása előtt.

Másik megoldás lehet, hogy egyszerűen elfelejtkezünk a ReportSmithről, és QuickReportot, vagy valamilyen más VCL komponenst használjuk. Ahogy azt a QrNav példában láthattuk, az ilyen típusú jelentésekhez nagyon egyszerűen adhatunk összegeket. A QuickReport komponenssel készíthetünk tételcsoportokat, és csoportonként is számíthatunk összegeket, így előállítva egy, a ReportSmith jelentéséhez nagyon hasonló dolgot, csak sokkal kevesebb munkával.

Ehhez a próbához elővettük a tizenhatodik fejezet Calc példáját, és elkészítettük belőle az új, QrCalc változatot. Az új jelentés a 28.15. ábrán látható jelentés formon alapul, ami négy sávból áll: a címsor (rbTitle), egy fejlécsávként használt QRGroup (csoport), egy közönséges sáv (rbDetail), és egy, a QRGroup komponenssel összekapcsolt csoport lábléc sáv (rbGroupFooter). A csoport komponens a jelentés része, és az elemeket az Expression tulajdonságának megfelelően csoportosítja, ebben az esetben ez Continent, hogy e mező alapján történjen a csoportosítás.



28.15. ábra *Az OrCalc példa QuickReport formja tervezés közben*

A különböző sávoknak különböző címke (QRLabel), adatbázis szöveg (QRDBText), és számított (QRExpr) komponensei vannak. A részleteket tartalmazó sávon vannak a mezők értékei, és a fejlécben a kontinens neve néhány más értékkel együtt. A lábléc a csoportba tartozó tételek számát, valamint az össznépesség értékét tartalmazza. A részleteket a példa formján láthatjuk, ami természetesen rajta van a CD-n, valamint a pontosabb részletekért célszerű a szöveges leírását megnézni.

Nagyon egyszerű ehhez hasonló QuickReport jelentést készíteni, de azt nem tudnánk hozzákötni a COUNTRY.DB táblához, mert a tábla név szerint van lerendezve, ami miatt sok, alig néhány tételes csoportunk lenne, amikből kontinensenként is több lenne, ugyanis a QRGroup komponens mindig új csoportot kezd, amikor a figyelt mező értéke megváltozik.

Erre megoldásként létrehozhattunk volna egy másodlagos indexet a Paradox táblára, ehelyett azonban egy Query (lekérdezés) komponensen keresztül fogjuk elérni az adatokat, amiben egy belső SQL paranccsal rendezzük le az országokat kontinensek szerint:

```
select * from country
  order by continent, name
```

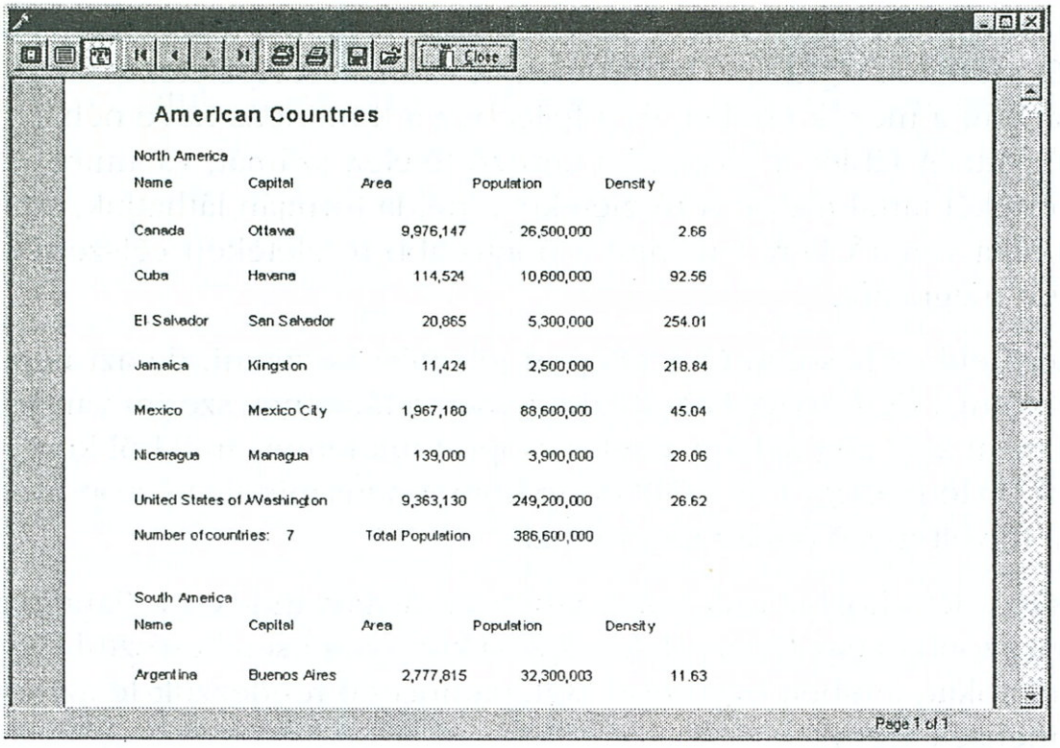
Ezután a változtatás után meg kell határoznunk a lekérdezés mezőit, hozzá kell adnunk egy új számított mezőt (népsűrűség), és módosítanunk kell az OnCalcFields esemény kezelőjét, hogy a tábla mezői helyett inkább a lekérdezésünkre hivatkozzon. A kódot magát a CD-n találhatjuk meg. Ez nagyon hasonló a Calc3 példa programjára, csak most Table komponens helyett Query-t használunk.

És mi lett mindennek a végeredménye? Most már a QuickReport komponensek segítségével érhetjük el a számított mezőket, mint a Query bármely más mezőjét. A ReportSmith megoldáshoz képest ez jelentősen egyszerűbb és gyorsabb, a végeredmény, pedig szinte teljesen megegyezik, ahogy az a 28.16. ábrán látható.

Valójában használhattunk volna másik, talán jobb, megoldást is, hiszen mivel SQL-t használtunk az adatok kinyerésére, akár a lekérdezésben is elkészíthettük volna a számított mezőnket, valahogy így:

```
select country.*, (Population / Area) as Density
from country
order by continent, name
```

A kód azért maradt ilyen, mert a példa egyik célja az volt, hogy megmutassuk, hogy hogyan lehet testreszabni egy ReportSmith jelentést BASIC programmal, egy QuickReport jelentést pedig Object Pascal nyelven. Mindemellett, az SQL nagyon korlátozott, ha valami, akár csak kicsivel is összetettebbet kell kiszámítanunk, emiatt gyakran amúgy is más nyelveket kell használnunk.



28.16. ábra *A QrCalc példa csoportos listája, ami QuickReport komponensekkel készült*

Hogyan tovább?

Ebben a fejezetben megnéztük a Delphi nyomtatási lehetőségeit. Láttuk, hogy milyen egyszerű kinyomtatni egy form képét, vagy egy szövegállományt hozzárendelni a nyomtatóhoz, és arra írni. A Printer objektum Canvas tulajdonságával mindenféle, akár nagyon összetett nyomtatási kép is elkészíthető. Delphiben a vászonnal dolgozunk, mind a képernyő, mind a nyomtató kimenet esetében. Ez azt jelenti, hogy a két művelet akár közös kódot is használhat, így valódi WYSIWYG (amit látsz, azt kapod – What You See Is What You Get) alkalmazást készítve, ahogy azt a Shape6 példán láthattuk. Adatbázis formok kinyomtatására három teljesen különböző megközelítés szolgál: közvetlenül írhatunk a nyomtatóra, használhatjuk a Delphi QuickReport komponenseit, vagy igénybe vehetjük a Client/Server Delphi mellé adott ReportSmith alkalmazást a bonyolultabb jelentések elkészítéséhez. A három technika eltérő lehetőségeket biztosít, de talán a legkönnyebben elkészíthető a QuickReportot használó megoldás.

Miután megbeszéltük, hogy hogyan tudunk programból nyomtatni, áttérhetünk egy másik nagyon fontos területre, az adatok állományba mentésére. A valóságban, akár adatbázisokkal foglalkozó, akár más Delphi programokban gyakran szükség lesz arra, hogy információkat eltároljunk a program következő indításáig.

29. FEJEZET

Fájlműveletek az alkalmazásokban

E fejezet tartalma:

- A fájlok és a Pascal nyelv
- Fájlkezelés a Delphi komponensekben
- Fájlrenszer komponensek
- Az adatfolyamok

Fájlműveletek az alkalmazások készítésénél

Programok készítése során az adatok fájlokba mentése, illetve visszatöltése a legtöbb esetben lényeges feladatnak számít. Már számos példát láttunk arra, hogyan tudunk a Delphiben adatokat elmenteni és visszatölteni, és számos alkalmazását láttuk a beépített fájlmegnyitási és elmentési párbeszédablakoknak. Bár a legtöbb ilyen lehetőséget már megismertük, azért a Delphi tartalmaz még néhányat.

A Delphi fájlművelet-támogatása tulajdonképpen három területre osztható:

- Az Object Pascal nyelvben található nyelvi elemekkel – a `file` kulcsszó és egy adattípus használatával, valamint számos függvénnyel a System egységben.
- VCL fájl támogatás – `TStream` és `TComponent` és a `TIniFile` osztályok segítségével, valamint számos komponensben vannak fájlbetöltési és elmentési metódusok.
- Adatbázis támogatás – itt most csak a fájlszintű adatbázis-támogatásról van szó, úgymint dBase és Paradox táblák.

Ebben a fejezetben a két első megközelítést fogjuk megvizsgálni részletesen, az adatbázisokról a 16. és 17. fejezet szól.

A fájlok és a Pascal nyelv

Az egyik sajátossága a Pascal nyelvnek más nyelvekkel szemben az, hogy beépített fájlkezelést tartalmaz. Mint ahogy az a 4. fejezetből kiderült, a nyelvben található egy `file` kulcsszó, amely egy speciális típus, ugyanúgy, mint a tömb vagy a rekordtípus. A `file` kulcsszóval új típust definiálunk, majd az új típus felhasználható új változók deklarálásához:

```
type
  IntFile: file of Integers;
var
  IntFile1: IntFile;
```

A `file` kulcsszó használható a típus megadása nélkül is, ekkor az ilyen típusú változók tipizálatlan fájlokra fognak hivatkozni. Még egy alternatív lehetőség: használható a System egységben deklarált TextFile típus, amely az ASCII karakterekből álló fájlok egyik lehetséges típusa. Minden fájltípusnak megvannak a saját előredefiniált rutinjai, ahogy azt majd később a fejezet során látni fogjuk. Ha deklaráltunk egy fájlváltozót, akkor hozzárendelhetjük egy fizikailag létező fájlhoz az AssignFile metódus használatával. A következő lépés rendszerint a Reset metódus hívása, ha olvasásra nyitottuk a fájlt, illetve a Rewrite abban az esetben, ha írni akarjuk. A Reset használatával is tudunk írni a fájlba, a két megnyitás között az a különbség, hogy a Rewrite fizikailag törli

az általunk megadott nevű fájlt, létrehozza újra üresen és megnyitja. Az Append metódussal tudunk új rekordokat felvenni a fájl végére, anélkül, hogy a régiek pozíciója, illetve tartalma sérülne. Ha az összes tervezett ki- és bemeneti műveletet végrehajtottuk a fájlt le kell zárni a CloseFile metódus hívásával. Ezek a műveletek általában általában egy try-finally blokkban vannak, kezelve ezzel a felmerülő hibákat.

A Delphiben más fájlkezelő rutinok is vannak, amint az az alábbi listában is látszik:

```
Append                  FileClose               Flush
AssignFile              FileCreate              GetDir
BlockRead               FileDateToDateTime      IOResult
BlockWrite              FileExists              MkDir
ChangeFileExt           FileGetAttr             Read
CloseFile               FileGetDate             Readln
DateTimeToFileDate      FileOpen                Rename
DeleteFile              FilePos                 RenameFile
DiskFree                FileRead                Reset
DiskSize                FileSearch              Rewrite
Eof                     FileSeek                RmDir
Eoln                    FileSetAttr             Seek
Erase                   FileSetDate             SeekEof
ExpandFileName          FileSize                SeekEoln
ExtractFileExt          FileWrite               SetTextBuf
ExtractFileName         FindClose               Truncate
ExtractFilePath         FindFirst               Write
FileAge                 FindNext                Writeln
```

Ezek közül nem mindegyik része az eredeti Pascal nyelvnek, de sokuk már jó ideje benne van a Borland Pascalban. Részletesebb információ ezekről a rutinokról a Delphi Súgó fájljaiban találhatunk. Itt most csak három egyszerű példát fogok bemutatni annak demonstrálására, hogy hogyan kell használnunk ezeket a rutinokat.

Szövegfájlok kezelése

Az egyik legáltalánosabban használt fájlformátum a szövegfájl formátum. Mint ahogy azt már az előbb említettem, a Delphi speciálisan kezeli a szövegállományokat, ennek jelentőségét mutatja a külön típus (TextFile) a System egységben. Az előző fejezetben úgy nyomtattunk ki egy szövegállományt, hogy a hozzá tartozó változót egyszerűen átadtunk az AssignPrn függvénynek. Hasonló módon, bármely fájlba irányíthatunk szöveget, csak a megfelelő fájlváltozóhoz hozzá kell azt rendelni.

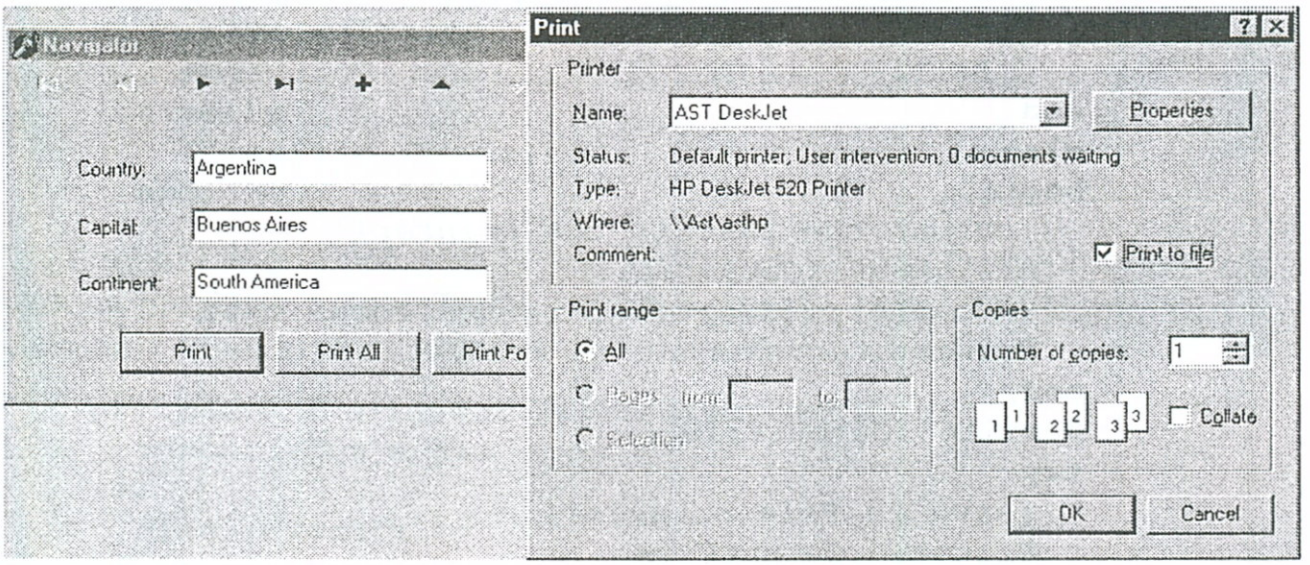
Ezért az első példánk a 28. fejezetben lévő PrintNav alkalmazás továbbfejlesztése lesz. Az új verzió neve PrnNav2 lesz, a form ugyanaz lesz, bővítve egy

PrintDialog és egy SaveDialog komponenssel. A Print to File jelölőnégyzet állapota fogja mutatni a programunknak, hogy a szöveget fájlba kell-e menteni, vagy nyomtatni kell. Hogy ez megjelenjen, egyszerűen állítsuk be a PrintDialog komponens Options tulajdonságának poPrintToFile kapcsolóját igazra.

Ebben az új verzióban, az Open gomb megnyomására a nyomtatási párbeszédablak fog megjelenni (a Print To File jelölőnégyzettel együtt, ahogy az a 29.1. ábrán is látható), ahelyett, hogy a nyomtatás rögtön elkezdődne. Tesztelhetjük, hogy a Print to File be van-e kapcsolva azzal, hogy megnézzük a Print párbeszédablak PrintToFile tulajdonságának értékét. Ez esetben a program a szövegállományt nem a nyomtatóra küldi, hanem a Save párbeszédablak segítségével bekér egy fájlnevet és a szöveget abba írja ki.

A kulcsművelet a szövegállomány hozzárendelése a fájlváltozóhoz:

```
AssignFile (PrintFile, SaveDialog1.FileName);
```



29.1. ábra *A PrnNav2 példa Print párbeszédablaka. Figyeljük meg, hogy a Print To File jelölőnégyzet a PrintDialog megfelelő opciójával aktiválva van*

Ezután tudunk írni a fájlba a Rewrite eljárás meghívása után:

```
Rewrite (PrintFile);
Writeln (PrintFile, Label1.Caption, ' ', DbEdit1.Text);
Writeln (PrintFile, Label2.Caption, ' ', DbEdit2.Text);
CloseFile (PrintFile);
```

Mint ahogy azt az előbb említettem, a folyamat biztonságosabbá tétele érdekében érdemes az output műveleteket egy `try` blokkba tenni, a CloseFile hívást pedig egy `finally` blokkba, így bármilyen input-output hiba kivédhető.

Továbbá, mivel az kimenetre küldés kódrészlete duplikáltan jelenik meg a forrásban (a nyomtatás és fájlba írás esete hasonló), átírjuk a kódot úgy, hogy ez a kódrészlet csak egyszer szerepeljen benne, mint ahogy az a PrintButtonClick metódus új forrásának listájában az alábbiakban látható:

```
procedure TNavigator.PrintButtonClick(Sender: TObject);
var
  OutputFile: TextFile;
begin
  {show the Print dialog box}
  if PrintDialog1.Execute then
  begin
    {if the Print to File check box is selected}
    if PrintDialog1.PrintToFile then
    begin
      {choose a file name}
      if SaveDialog1.Execute then
        {output the text to a file}
        AssignFile (OutputFile, SaveDialog1.FileName)
      else
        Exit;
    end
    else
    begin
      {output the text to the printer}
      AssignPrn (OutputFile);
      {set the printer font to the form font}
      Printer.Canvas.Font := Font;
    end;
    Rewrite (OutputFile);
    try
      {save to file or print}
      Writeln (OutputFile,
          Label1.Caption, ' ', DBEdit1.Text);
      Writeln (OutputFile,
          Label2.Caption, ' ', DBEdit2.Text);
      Writeln (OutputFile,
          Label3.Caption, ' ', DBEdit3.Text);
    finally
      CloseFile (OutputFile);
    end;
  end;
end;
```

A másik gomb a Print All, amelynél bonyolultabb output műveletekre van szükség, ezért úgy döntöttem, hogy itt némileg más megközelítést alkalmazok. Olyan eljárásról van szó, amely egy adatbázistáblát nyomtat, illetve ír egy szövegfájlba. Álljon itt a kód lényeges része, amely a szokásos `while` ciklust

használva végigmegy az összes rekordon és további két `finally` blokkban végzi az írást (ezek közül az egyik az alábbi listából hiányzik):

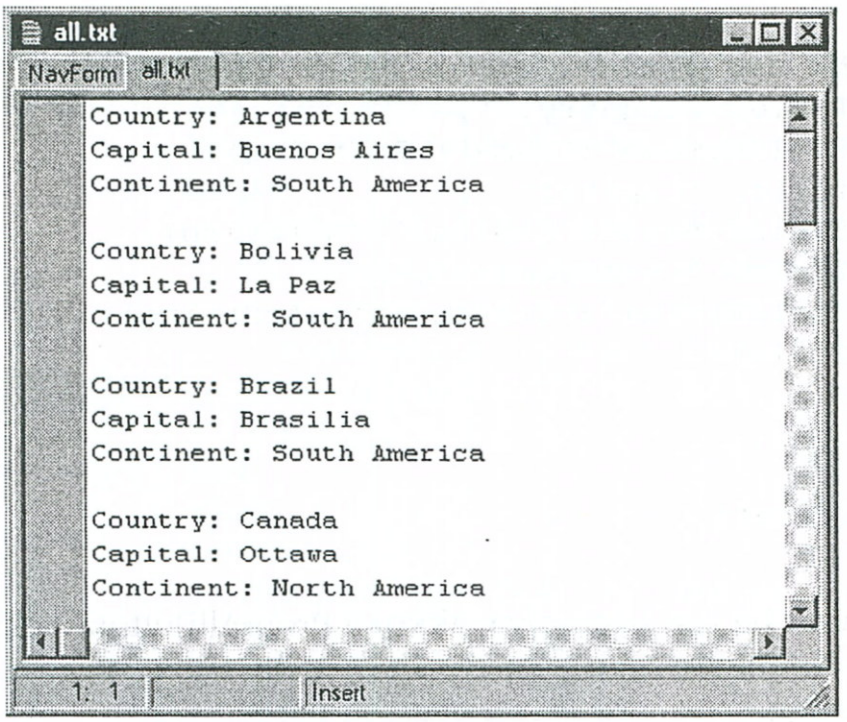
```
procedure TNavigator.TableToFile (var TFile: TextFile);
begin
  ...
  Rewrite (TFile);
  try
    Table1.First;
    while not Table1.EOF do
    begin
      {output the two fields, and a blank line}
      Writeln (TFile, 'Country: ',
        Table1.FieldByName ('Name').AsString);
      Writeln (TFile, 'Capital: ',
        Table1.FieldByName ('Capital').AsString);
      Writeln (TFile, 'Continent: ',
        Table1.FieldByName ('Continent').AsString);
      Writeln (TFile);
      Table1.Next;
    end;
  finally
    CloseFile (TFile);
  end;
```

Ha a Print to File jelölőnégyzet be van kapcsolva, akkor a ProntAllButtonClick metódus meghívja a TableToFile metódust, amely az aktuális fájlba, vagy a nyomtatóra küldi ki az adatokat:

```
procedure TNavigator.PrintAllButtonClick(
  Sender: TObject);
var
  File1: TextFile;
begin
  if PrintDialog1.Execute then
    if PrintDialog1.PrintToFile then
    begin
      if SaveDialog1.Execute then
      begin
        {assign the output to a real file}
        AssignFile (File1, SaveDialog1.FileName);
        TableToFile (File1);
      end;
    end
    else
    begin
      {assign the printer to a file}
      AssignPrn (File1);
      {set the font of the form, and output the file}
```

```
            Printer.Canvas.Font := Font;
            TableToFile (File1);
        end;
    end;
```

Futtassuk a példaprogramot, mentsük el a táblát egy szöveges állományba, és nyissuk meg az eredményként kapott fájlt (például a Delphi editorában, ahogy az a 29.2. ábrán látható).



29.2. ábra *A PrnNav2 példaprogram által előállított szövegfájl, ha az összes rekord nyomtatását választottuk*

Egy szövegkonverter

A fájlkezeléshez kapcsolódó első példánkban egy szövegfájlt állítottunk elő, amely egy adatbázis tábla tartalmát használta fel. A következő példában egy létező fájlt fogunk feldolgozni, és létrehozunk egy újat, az elsőhöz képest megváltozott tartalommal. Filter nevű programunk konvertálni fogja szövegünket úgy, hogy abban csak a mondatok első betűi legyenek nagybetűsek.

A form két csak olvasható szerkesztőmezőt tartalmaz az input és output fájl nevének megjelenítésére, valamint két gombot a fájlok kiválasztására, ezek a műveletek a beépített párbeszédablakok segítségével fognak végrehajtódni. A form alsó része tartalmaz egy rádiócsoportot és egy bitmap gombot (ConvertBitBtn) a kívánt konverzió végrehajtására a kiválasztott fájlokon. A

rádiógomb-csoportnak három eleme van, mint ahogy az a form szöveges leírásából kiolvasható:

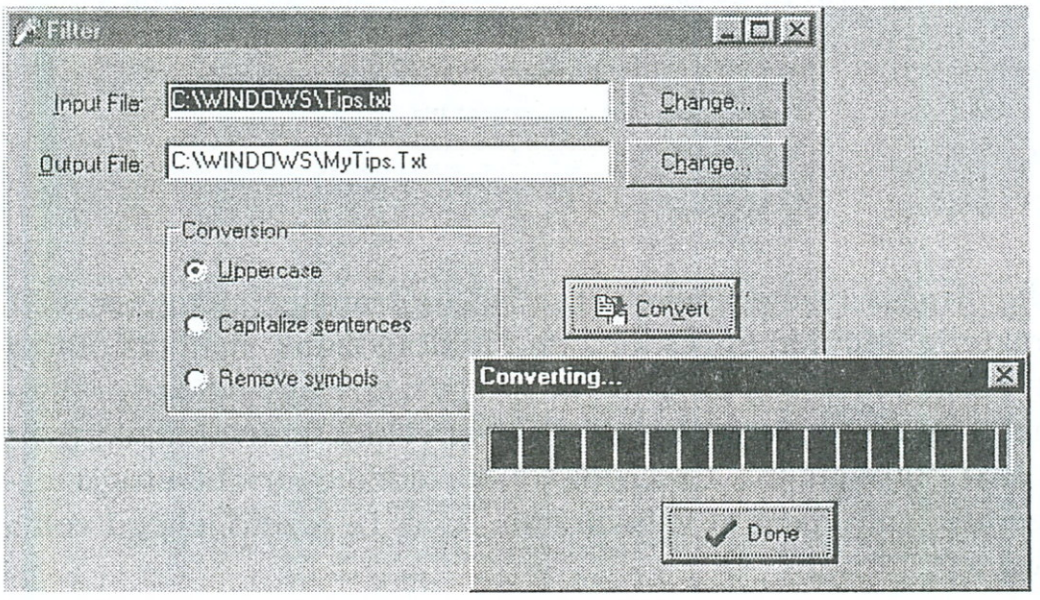
```
object RadioGroup1: TRadioGroup
  Caption = 'Conversion'
  Items.Strings = (
    '&Uppercase'
    'Capitalize &sentences'
    'Remove s&ymbols')
```

A felhasználó a két gombra kattintva kiválaszthatja az be- és kimeneti fájl nevét, és a két név megjelenik a két szerkesztőmezőben:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    Edit1.Text := OpenDialog1.Filename;
end;
```

A gomb aktiválja a SaveDialog1 párbeszédablakot. A példa tényleges programkódja a bitmap gomb OnClick eseményére meghívódó eljárás, amely aktiválja a három részből álló konverziós rutint. Ez a három hívás egy case szerkezetben van, a ConvertBitBtn gomb OnClick eseménykezelő metódusában:

```
case RadioGroup1.ItemIndex of
              0: ConvUpper;
              1: ConvCapitalize;
              2: ConvSymbols;
            end;
```



29.3. ábra *A konverziós eljárás frissíti a form folyamatsávját, mutatva ezzel a felhasználónak, hogy hol tart a fájl százalékos feldolgozása*

A teljes forrás megtalálható a CD-n. Mielőtt meghívódna bármelyik konvertáló eljárás, a ConvertBitBtnClick metódus megjelenít egy párbeszédablakot, rajta egy folyamatsávval, amely azt mutatja a felhasználónak, hogy hol tart a konverzió (mint az a 29.3. ábrán látható). Ez a metódus végzi a fájlkezelő feladatok nagy részét – ez nyitja meg az input fájlt bájt típusú fájlként (egyszerre egy bájtot tudunk olvasni belőle, illetve írni bele), így használható rá a FileSize eljárás, amelyet nem hívhatunk meg szövegfájlokra. Ezután lezárjuk a fájlt és újranyitjuk szövegfájlként.

Mivel a program két állományt nyit meg, és ezek közül a műveletek közül akármelyik lehet sikertelen, ezért két egymásba ágyazott try blokkot használunk, ezzel biztosítva a megfelelően magas szintű védelmet, bár azzal, hogy a felhasználó a fájlneveket a beépített fájl párbeszédablakokkal választja ki, már védettséget biztosítunk az állományok helyes kiválasztására. Most pedig vessünk egy részletesebb pillantást a konverziós rutinok egyikére. A legegyszerűbb a három közül a ConvUpper eljárás, amely minden karaktert a szövegben nagybetűssé alakít át. Íme a kód:

```
procedure TForm1.ConvUpper;
var
  Ch: Char;
  Position: LongInt;
begin
  Position := 0;
  while not Eof (FileIn) do
  begin
    Read (FileIn, Ch);
    Ch := UpCase (Ch);
    Write (FileOut, Ch);
    Inc (Position);
    ConvertForm.ProgressBar1.Position :=
      Position * 100 div FileLength;
    Application.ProcessMessages;
  end;
end;
```

Ez az eljárás addig olvassa a karaktereket a forrásfájlból, amíg el nem ér az EOF (End of file) karakterig. Minden egyes karaktert feldolgozunk, majd az output fájlba másolunk. Másik lehetőség az lehet, hogy egyszerre egy egész sor olvasunk be, majd azt a megfelelő stringkezelő függvényekkel konvertáljuk, és az eredményt kiírjuk az output fájlba. Ez a programot sokkal gyorsabbá fogja tenni. Ez a megközelítés, amit most itt választottam, csak bevezető jellegű volt a feladat megoldására.

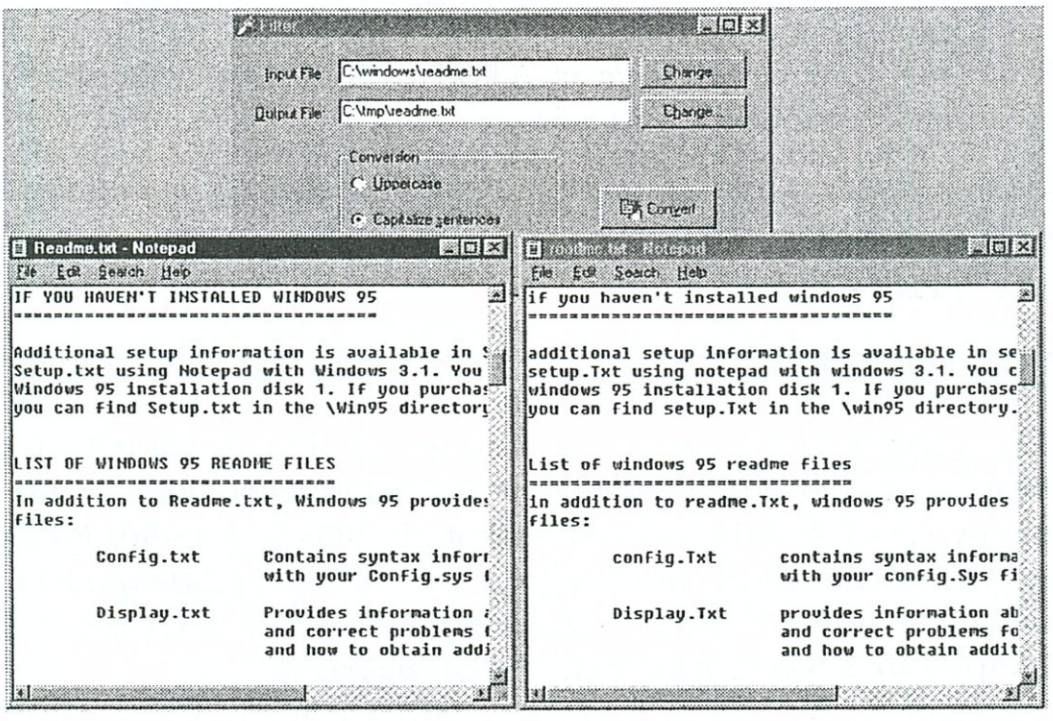
A konverziós rutin kódját még az nehezíti meg, hogy frissíteni kell a párbeszédablak folyamatsávját. A konverzió minden egyes lépésében növelni kell az aktuális pozícióval egy hosszú egész típusú változó értékét. Ezt a változót

fogjuk majd felhasználni arra, hogy megmutassuk, hol tart százalékosan a konverzió, ahogy az az alábbi kódban látható:

```
while not Eof (FileIn) do
begin
  Read (FileIn, Ch);
  if Ch < Chr (127) then
    Write (FileOut, Choose);
```

A mondat eleji szavak nagybetűssé tételét végző eljárás bonyolult, a teljes kód a CD-n található meg. A konverzió egy négyágú case szerkezeten alapul:

- Ha a karakter nagybetű, és egy mondatvégi írásjel után áll (ezt a Period logikai változó jelzi), akkor úgy marad, ahogy volt, egyébként kisbetűssé alakítjuk. Ezt az átalakítást nem beépített eljárással végezzük, egyszerűen azért, mert ilyen nem létezik egyszerű karakterekre. Erre a célra el kellett készítenem egy alacsonyszintű függvényt, melynek a LowCase nevet adtam.
- Ha a betű kisbetű volt, akkor csak akkor alakítjuk át nagybetűvé, ha az a mondat kezdőbetűje.
- Ha a karakter egy mondatvégi írásjel (pont, kérdőjel, vagy felkiáltójel), akkor a Period változó értékét igazra állítjuk.
- Ha a karakter bármi más, akkor egyszerűen átmásoljuk, és a Period értékét hamisra állítjuk.



29.4. ábra *A Filter példa első betűket nagybetűvé konvertáló eljárásának eredménye*

A 29.4. ábra mutat példát a kód hatására, egy szövegállományt láthatunk a konverzió előtt és után. A program messze van egy profi megoldástól, de ez csak az első lépés egy komoly konverziós program megírása felé. A legnagyobb hátránya a programnak, hogy a tulajdonneveket gyakran kisbetűsekké konvertálja, és bármely betűt, amely pont után áll, nagybetűre cserél (akkor is, ha például egy fájl kiterjesztéséről van szó).

Általános adatok mentése

A szövegfájlok használata mellett elmenthetünk más típusú adatokat is, úgymint egész és valós számokat, tömböket vagy rekordokat (objektumokat nem menthetünk el közvetlen módon. Ennek Delphi-beli támogatásáról a fejezet későbbi részében lesz szó). Egy saját fájltípus használata a szövegfájl típus helyett általában előnyösebb, mert kevesebb helyet foglal (egy szám szöveges ábrázolása nyilván több lemezterületet foglal, mint a bináris érték), és ez a megközelítés a felhasználók számára nehezebbé teszi a fájl szerkesztését, amely szintén előny lehet.

Az előző fejezet példáit vizsgálva találtam egy olyan programot, amelynek előnye származhat a bináris fájlkezelésből – egy olyan példáról van szó, amelynél célszerűbb egy egész számokból álló fájlt használni, mint egy szövegfájlt. A következő példánk (Graph3) a Graph alkalmazáson alapul, amelyet a 11. fejezetben készítettünk. A program egy TeeChart komponenst használ az összegyűjtött adatok táblázatos megjelenítésére. Mivel tulajdonképpen számok sorozatáról van szó, nincs értelme az adatokat szövegként menteni, bár a táblázat az értékeket stringekként tartalmazza. A példaform egy File és egy Help menüt tartalmaz az About menüponttal együtt. A fájlokkal kapcsolatos menüpontok legfontosabbak: az Open, a Save és a Save as. A kód további része csak csekély változáson esett át az új verzióban.

Hogyan mentsünk el egész számok sorozatát egy fájlba? Először is definiálni kell egy fájlt az alábbi módon:

```
SaveFile: file of Integer;
```

Majd a fájlváltozóhoz hozzá kell rendelni egy fizikailag létező fájlt, meg kell nyitni azt, és végre kell hajtani a megfelelő műveleteket egy `try` blokkon belül. A program használ két lokális mezőt, egy stringet az aktuális fájl nevének tárolására (CurrentFile) és egy logikai értéket, amely azt mutatja, hogy az adatok változtak-e vagy sem (Modified). Kezdéskor a Modified kapcsolót igazra állítjuk azokban a metódusokban, amelyeknek van hatása az adatokra (az Update gomb megnyomásával, vagy a kiválasztott grafikon típusának megváltoztatásával). A kapcsolót a SaveClick1 metódus használja:

```
procedure TForm1.Save1Click(Sender: TObject);
var
  SaveFile: file of Integer;
```

```
    I, J, Value: Integer;
begin
  if Modified then
    if CurrentFile = '' then
      {call save as}
      SaveAs1Click (self)
    else
    begin
      {save to the current file}
      AssignFile (SaveFile, CurrentFile);
      Rewrite (SaveFile);
      try
        {write the value of each grid element}
        for I := 1 to 5 do
          for J := 1 to 4 do
          begin
            Value := StrToIntDef (Trim (
              StringGrid1.Cells [I, J]), 0);
            Write (SaveFile, Value);
          end;
        Value := Integer (ChBoxMarks.Checked);
        Write (SaveFile, Value);
        Modified := False;
      finally
        CloseFile (SaveFile);
      end;
    end;
end;
```

Az adatok elmentéséhez a program a stringrács minden értékét (számmá konvertálásuk után) elmenti, utána menti a jelölőnégyzet értékét (a logikai Checked tulajdonság értékét szintén egész számmá konvertálja). Ehhez a program két egymásba ágyazott ciklust használ, végigmegy a teljes stringtáblázaton és utána a kombinált listában kiválasztott érték kódját hozzáfűzi a fájl végéhez. Figyeljük meg az ideiglenes Value változó használatát: a Write és Read eljárásoknak cím szerinti paraméterre van szükségük, így aztán nem lehet egy tulajdonságot átadni nekik, mert ezek memóriacíméhez közvetlenül nem férünk hozzá.

Természetesen az adatokat az elmentés sorrendjében kell visszaolvasni, mint ahogy az az Open1Click metódus forrásában látható:

```
procedure TForm1.Open1Click(Sender: TObject);
var
  LoadFile: file of Integer;
  I, J, Value: Integer;
begin
  if OpenDialog1.Execute then
```

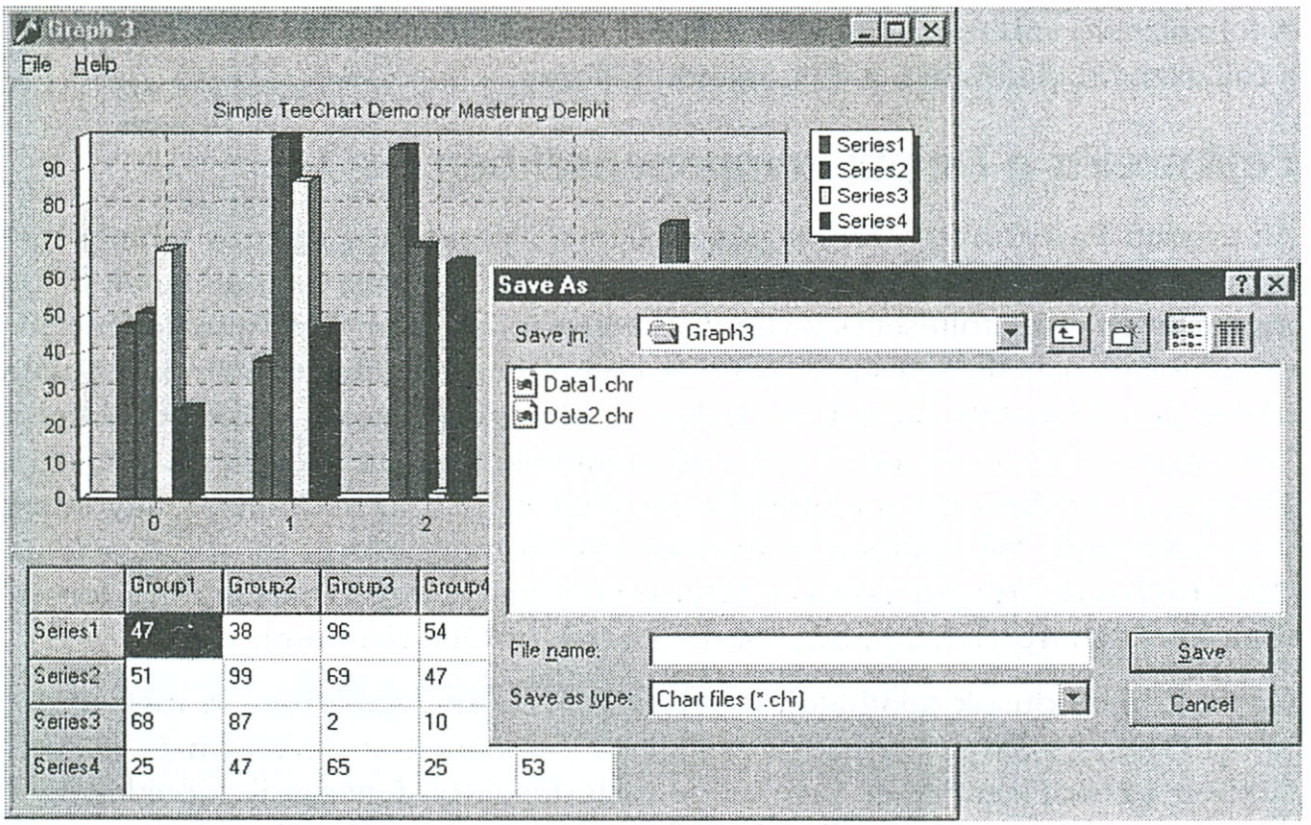
```
    begin
      CurrentFile := OpenDialog1.Filename;
      Caption := 'Graph 3 [' + CurrentFile + ']';
      {load from the current file}
      AssignFile (LoadFile, CurrentFile);
      Reset (LoadFile);
      try
        {read the value of each grid element}
        for I := 1 to 5 do
          for J := 1 to 4 do
          begin
            Read (LoadFile, Value);
            StringGrid1.Cells [I, J] := IntToStr(Value);
          end;
        Read (LoadFile, Value);
        ChBoxMarks.Checked := Boolean(Value);
      finally
        CloseFile (LoadFile);
      end;
      ChBoxMarksClick (self);
      UpdateButtonClick (self);
      Modified := False;
    end;
  end;
```

A fenti kód végén a program frissíti a kombinált listát (az aktuális kijelölést) és bemásolja az adatokat a rácsba (az Update gomb működését szimulálva). A csoport harmadik metódusa a File/Save as menüponthoz kapcsolódik:

```
  procedure TForm1.SaveAs1Click(Sender: TObject);
  begin
    if SaveDialog1.Execute then
    begin
      CurrentFile := SaveDialog1.Filename;
      Caption := 'Graph 3 [' + CurrentFile + ']';
      {call save}
      Modified := True;
      Save1Click (self);
    end;
  end;
```

Mint az látható, a Save1Click és a SaveAs1Click metódusok egymást hívják. A Save1Click meghívja a második metódust, hogy rákérdezzen a fájl nevére, amikor a fájl megváltozott és nincs még hozzárendelve fájlnév a CurrentFile változóhoz. Ez a string másolódik a form címsorába is minden alkalommal, amikor változás történik (akkor is ha betöltünk egy fájlt, és akkor is, ha az aktuálisat elmentjük egy új néven). SaveAsClick1 a visszatérésnél meghívja a Save1Click metódust a fájl új néven való elmentéséhez.

A fájlkezelés lezárásához be kell állítani a SaveDialog és OpenDialog komponensek tulajdonságait. Mindkettőnek van egy speciális filtere (*.chr), és néhány beépített opciója (a CHR fájlkiterjesztés saját találmány, csak ez a program használja, nem a TeeChart komponenshez tartozik). A 29.5. ábrán láthatjuk a Graph3 program egyik párbeszédablakának működését.



29.5. ábra *Fájl elmentése új néven a Graph3 példaprogramban*

A kód utolsó részeként be szeretném mutatni, hogyan kell korrekt ellenőrzést végezni, ha a felhasználó be akarja zárni a programot:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
var
  Code: Word;
begin
  if Modified then
  begin
    Code := MessageDlg (
      'Data has changed, do you want to save it?',
      mtConfirmation, mbYesNoCancel, 0);
    if Code = idYes then
      Save1Click (self);
    if Code = idCancel then
```

```
        CanClose := False
      else
        CanClose := True;
    end
    else
      CanClose := True;
  end;
```

A felhasználó válaszától függően a form bezárul illetve nyitva marad, valamint a felhasználó elmentheti a fájlt, mielőtt kilépne.

Fájlkezelés a Delphi komponensekben

Az eredeti Pascal nyelvi támogatás mellett a Delphi még számos lehetőséget biztosít a fájlok kezelésére. Számos komponens támogatja a tartalmának fájlba mentését és visszatöltését (szöveg illetve kép formájában), és vannak speciális osztályok is a fájlkezeléshez. Számos komponensosztálynak van SaveToFile illetve LoadFromFile metódusa. Ebben a könyvben ezeket a metódusokat a `TBitmap` , `TPicture` és `TStrings` osztályokra fogjuk alkalmazni (ezekre a `TMemo`, `TListbox` komponensekben és egyéb más komponensosztályokban van szükség). De elérhetők néhány adatkomponensre is, mint például `TBlobField`, `TMemoField` és `TGraphicField`, egyéb más grafikai formátumokra (`TGraphic`, `TIcon`, `TMetaFile`), és OLE tárolókra.

Hasonló metódusok találhatók a `TMediaPlayer` osztályban. A metódusok nevei Open és Save, bár ezek némileg különböznek szintaxisban és jelentésükben a LoadFromFile és SaveToFile metódusoktól. Egy másik fájlokkal kapcsolatos osztályt már használtunk a 25. fejezetben, ez a TIniFile volt. Ez az osztály speciálisan a Windows inicializációs fájljaira használható, vagy bármilyen más egyéb fájltípusra, amely ugyanezt a formátumot használja. A következőkben a fájlrendszer komponensekkel fogunk megismerkedni: A streamekkel, és olyan komponensekkel, amelyek megvalósítják az objektum perzisztenciát.

Fájlrendszer komponensek

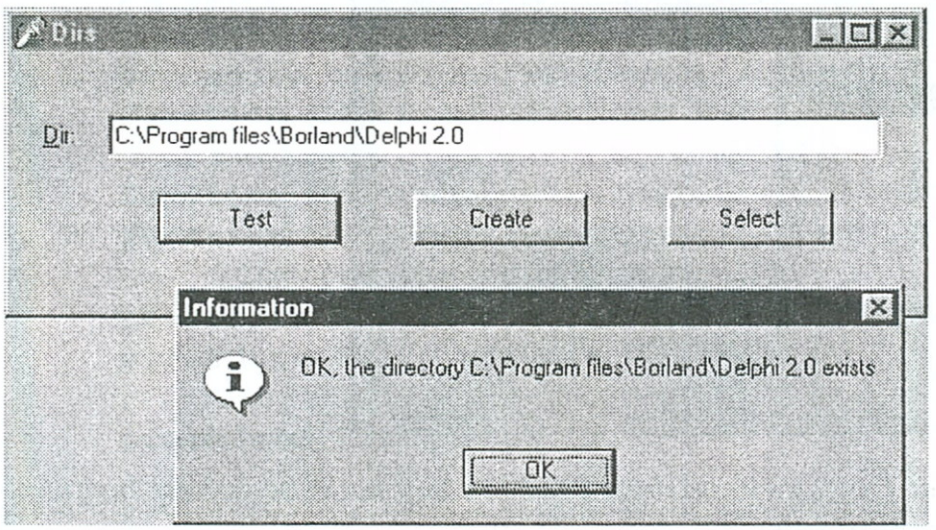
A Delphi fájlkezelést támogató fájlrendszer komponensei a komponenspaletta System lapján találhatók meg. Ezek a `TDirectoryListBox`, `TDriveComboBox`, `TFileListBox` és a `TFilterComboBox`. Ezeket az összetevőket használtuk a 12. fejezet TabOnly2 nevű példájában. Ezek a komponensek számos Delphi programozó számára jól ismertek. Ami kevésbé ismert, az az, hogy ugyanaz a FileCtrl egység, amely ezeket a komponenseket definiálja, tartalmaz három érdekes rutint is:

- DirectoryExists, amely egy könyvtár létezésének vizsgálatára szolgál.
- ForceDirectories, létrehoz több könyvtárat egymás alatt egy utasítással.

- SelectDirectory, egy előre definiált Delphi párbeszédablakot jelenít meg könyvtárválasztáshoz.

A következő példánk e három kevésbé ismert rutin működését mutatja be. A példaprogramot Dirs-nek hívják, van benne egy egyszerű form egy szerkesztőmezővel és három gombbal, ahogy az a 29.6. ábrán látható. A két első gomb fejlesztési időben nincs engedélyezve, csak akkor lesznek kiválaszthatók, ha a szerkesztőmezőben megjelenik valamilyen szöveg:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  if Edit1.Text <> '' then
  begin
    TestButton.Enabled := True;
    CreateButton.Enabled := True;
  end
  else
  begin
    TestButton.Enabled := False;
    CreateButton.Enabled := False;
  end;
end;
```



29.6. ábra *A Dirs példaprogram eredménye, ha az aktuális könyvtár létezik*

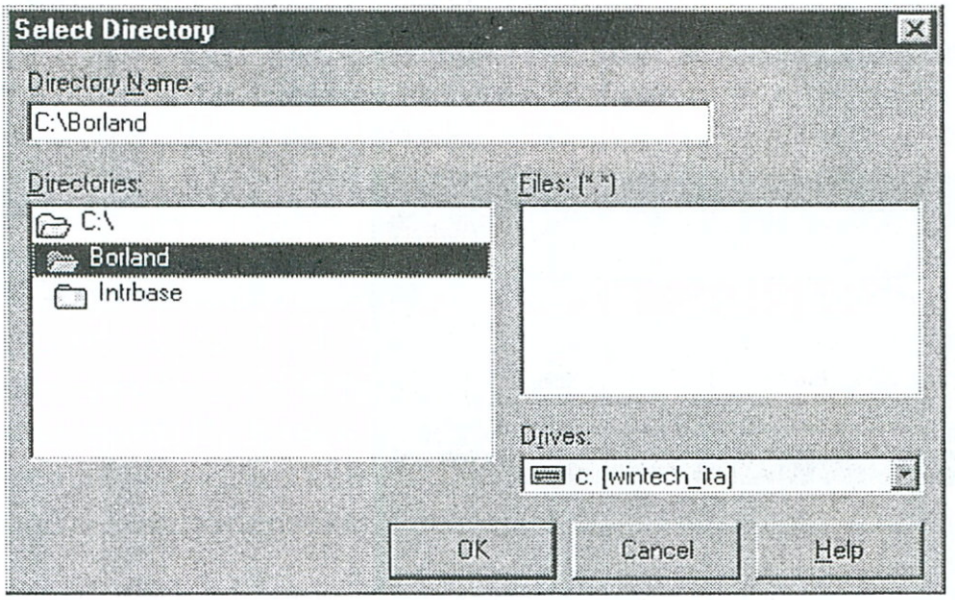
Az első gombhoz csatolt metódus megvizsgálja az aktuális könyvtár létezését, és létrehozza a megfelelő üzenetet (lásd 29.6. ábra):

```
procedure TForm1.TestButtonClick(Sender: TObject);
begin
  if DirectoryExists (Edit1.Text) then
```

```
      MessageDlg ('OK, the directory ' +
        Edit1.Text + ' exists', mtInformation, [mbOk], 0)
    else
      MessageDlg ('Sorry, the directory ' + Edit1.Text +
        ' doesn''t exist', mtError, [mbOk], 0);
end;
```

A második gomb megnyomásával a felhasználónak jóvá kell hagynia a műveletet, ezután létrejön a könyvtár (vagy legalábbis megpróbálja a Delphi létrehozni, ha a megadott útvonal rossz, akkor természetesen nem fog neki sikerülni):

```
procedure TForm1.CreateButtonClick(Sender: TObject);
begin
  if MessageDlg ('Are you sure you want to create the'+
      Edit1.Text + ' directory', mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
    ForceDirectories (Edit1.Text);
end;
```



29.7. ábra *A Select Directory párbeszédablak egy kevésbé ismert beépített párbeszédablak*

Az utolsó gomb OnClick eseménykezelője (ez a gomb mindig engedélyezett) egyszerűen megjeleníti a Delphi könyvtárválasztó párbeszédablakát, ahogy az a 29.7. ábrán látható. A három kapcsoló a SelectDirectory függvényhez van kötve, lehet új könyvtárat létrehozni, kérve a felhasználó jóváhagyását.

```
procedure TForm1.SelectButtonClick(Sender: TObject);
var
  Text: String;
```

```
begin
  Text := Edit1.Text;
  if SelectDirectory (Text, [sdAllowCreate,
       sdPerformCreate, sdPrompt], 0) then
    Edit1.Text := Text;
end;
```

Az adatfolyamok

Egy másik érdekes fejezete a Delphi fájlkezelésének a folyamok (streamek) használata. A VCL definiálja az absztrakt `TStream` osztályt és annak három alosztályát: `TFileStream`, `THandleStream` és `TMemoryStream`. A `TStream` szülőosztály csupán néhány tulajdonsággal rendelkezik, de van néhány érdekes metódusa, amelyek az adatok elmentésére és visszatöltésére használhatók.

A metódusok széles skálájának leírása megtalálható a Delphi súgófájljaiban, de nem mindig könnyű megtalálni őket. A nekünk kellő információ megtalálásához, célszerű egyenként megkeresni őket név szerint a Delphi Súgójában. A következő metódusokról van szó:

```
function Read (var Buffer; Count: LongInt): LongInt;
function Write (const Buffer; Count: LongInt): LongInt;
function Seek (Offset: LongInt; Origin: Word): LongInt;
procedure ReadBuffer (var Buffer; Count: LongInt);
procedure WriteBuffer (const Buffer; Count: LongInt);
function CopyFrom (Source: TStream; Count: LongInt):
    LongInt;
function ReadComponent(Instance: TComponent):TComponent;
function ReadComponentRes(Instance: TComponent):
    TComponent;
procedure WriteComponent (Instance: TComponent);
procedure WriteComponentRes (const ResName: string;
    Instance: TComponent);
procedure ReadResHeader;
```

Számos függvénynek ezek közül csak komponensek készítésekor van jelentőssége, de néhányat, például a ReadBuffer vagy a WriteBuffer függvényeket akárki könnyen használhatja.

A `TStream`-ből nincs értelme új objektumpéldányt létrehozni, mert az osztály tulajdonképpen nem támogatja az adatok elmentését. Ehelyett inkább használjunk fájl alapú folyamokat, amelyek két fájl közti adatmozgást tesznek lehetővé. A `THandleStream` használatos, ha van egy Windows kezelőnk a fájlra. A `TFileStream`-et akkor alkalmazzuk, ha csak a fájl nevével rendelkezünk. Mindkét osztálynak speciális Create metódusa van, a fájl információk átvétele miatt. A harmadik osztály a `TMemoryStream`, amely a memóriába dolgozik és nem az aktuális fájlba. Speciális metódusaink vannak adatok átmásolására

vagy áttöltésére egy másik folyamra (vagy folyamról), amely lehet egy fájl is. Létrehozni és használni egy fájlfolyamot nagyon egyszerű: definiálni kell egy változót a TStream-ből leszármaztatott típussal:

```
var
  S: TFileStream;
begin
  if OpenDialog1.Execute then
  begin
    S := TFileStream.Create (OpenDialog1.FileName,
      fmOpenRead);
    try
      {use the stream S ...}
    finally
      S.Free;
    end;
  end;
end;
```

Ahogy a kódból is látható, a fájlstream Create metódusának két paramétere van: a fájl neve és a kért művelethez tartozó kapcsoló. Ebben az esetben olvasni kívánjuk a fájlt (fmOpenRead). Most már használhatjuk a folyamokat az eddigi fájlkezelés helyett, bár használatuk első ránézésre nem triviális. Nagy előnye a folyamoknak például az, hogy dolgozhatunk memóriába és menthetünk onnan fájlba, és ugyanez működik visszafelé is. Így jelentősen javulhat egy fájlorientált program sebessége. A következő kódrészlet azt mutatja be, hogyan tudunk egy fájlmásoló függvényt írni folyamok használatával (itt is két `try-finally` blokkot kellene alkalmazni a teljes védelemhez):

```
procedure CopyFile (SourceName, TargetName: String);
var
  Stream1, Stream2: TFileStream;
begin
  Stream1 := TFileStream.Create(SourceName,fmOpenRead);
  Stream2 := TFileStream.Create (TargetName,
      fmOpenWrite or fmCreate);
  Stream2.CopyFrom (Stream1, Stream1.Size);
  T.Free;
  S.Free;
end;
```

További fontos használata a folyamoknak (a fájl és memóriafolyamoknak egyaránt) adatbázisok BLOB típusú mezőinek, valamint más nagyobb méretű mezők közvetlen kezelése. Azaz, adatok elmentése illetve betöltése ilyen mezőknél egyszerűen úgy történik, hogy meghívjuk a megfelelő metódust.

Számfolyamok

A folyamok használatára példaként a Graph3 példát fogjuk átírni, folyamokat használva a sima fájlok helyett. Mindkét program hatása ugyanaz. Ez az új példa egy nagyon egyszerű megközelítést használ: létrehoz egy `TFileStream` típusú objektumot az adatok elmentéséhez és visszatöltéséhez, és használni fogja a `TStream` osztály Write és Read metódusait. Mindkét metódus paraméterként kap egy csomó adatot (egy adatbuffert) és az adatok méretét. Mivel egész számokkal dolgozunk, egyszerűen használhatjuk a `sizeof(integer)` kifejezést.

Az új Save1Click metódus kódja a következő (a folyamokra építve):

```
procedure TForm1.Save1Click(Sender: TObject);
var
  SaveStream: TFileStream;
  I, J, Value: Integer;
begin
  if Modified then
    if CurrentFile = '' then
      {call save as}
      SaveAs1Click (self)
    else
    begin
      {save to the current file}
      SaveStream := TFileStream.Create (
        CurrentFile, fmOpenWrite or fmCreate);
      try
        {write the value of each grid element}
        for I := 1 to 5 do
          for J := 1 to 4 do
          begin
            Value := StrToIntDef (Trim (
              StringGrid1.Cells [I, J]), 0);
            SaveStream.Write (Value, sizeof(Integer));
          end;
        Value := Integer (ChBoxMarks.Checked);
        SaveStream.Write (Value, sizeof (Integer));
        Modified := False;
      finally
        SaveStream.Free;
      end;
    end;
end;
```

Összehasonlíthatjuk ezt a metódust a Chart2 példa megfelelő metódusával, amelyet az előző fejezetben mutattunk be. Hasonlóan, az Open1Click metódust is frissítenünk kell a folyamhasználattal:

```
procedure TForm1.Open1Click(Sender: TObject);
var
  LoadStream: TFileStream;
  I, J, Value: Integer;
begin
  if OpenDialog1.Execute then
  begin
    CurrentFile := OpenDialog1.Filename;
    Caption := 'Chart [' + CurrentFile + ']';
    {load from the current file}
    LoadStream := TFileStream.Create (
      CurrentFile, fmOpenRead);
    try
      {read the value of each grid element}
      for I := 1 to 5 do
        for J := 1 to 4 do
        begin
          LoadStream.Read (Value, sizeof (Integer));
          StringGrid1.Cells [I, J] := IntToStr(Value);
        end;
      LoadStream.Read (Value, sizeof (Integer));
      ChBoxMarks.Checked := Boolean(Value);
    finally
      LoadStream.Free;
    end;
    ChBoxMarksClick (self);
    UpdateButtonClick (self);
    Modified := False;
  end;
end;
```

Komponensfolyamok

Különösen érdekes tulajdonsága a folyamoknak, hogy képesek kezelni a komponenseket is. Minden VCL komponens osztály alosztálya a `TPersistent` osztálynak (speciális osztály objektumok mentésére egy folyamba) és tartalmaznak metódusokat az összes tulajdonság és publikus mező elmentésére és visszatöltésére. Ezért minden a `TComponent`-ből leszármaztatott osztály el tudja menteni saját magát egy streamre, vagy létrehozható úgy automatikusan, hogy egy streamről töltődik be. Egy program a folyam WriteComponent és ReadComponent metódusait használva teheti ezt meg, ahogy azt majd a következő, Cref2 című példa során látni fogjuk.

Ahogy a név is sugallja, ez a példaprogram egy korábbi program, a ClassRef második verziója, amely a 6. fejezetből való. Az eredeti programban a felhasználó háromféle típusból választva számos új komponenspéldányt hozhat léte egy formon belül. Az új verzióban a felhasználó el is mentheti a komponenseket egy fájlba, és vissza is töltheti azokat.

Megjegyzés

A folyamok további részletes tárgyalása megtalálható a Delphi Developer's Handbook című könyvben található, mely tartalmazza az ebben a fejezetben ismertetett Cref2 példaprogram továbbfejlesztett változatát is.

A Cref2 formja egészen egyszerű. Egy panel van rajta három rádiógombbal. Mint a régi verzióban, a példaprogram osztályreferenciákat fog használni annak meghatározására, hogy milyen típusú komponenst kell létrehozni, olyankor, amikor a felhasználó kattint a form területén. A kódnak ez a része hasonló az előző verzióhoz. Az új dolgok a programban a menüpontokhoz rendelt metódusokban vannak: a New, Open, SaveAs, Exit és About pontoknál.

A New1Click metódus törli az összes létező komponenst, kivéve az eszközsorként használt panelt. Ehhez végignézi a form Controls tömbjét visszafelé (`downto`). Azaz, minden alkalommal, amikor egy új komponenst eltávolítunk, a Controls tulajdonság meg fog változni. Figyeljük meg, hogy ez a metódus beállítja a komponens Counter értékét is:

```
procedure TForm1.New1Click(Sender: TObject);
var
  I: Integer;
begin
  {delete all existing components, except the panel}
  for I := ControlCount - 1 downto 0 do
    if Controls.ClassName <> 'TPanel' then
        Controls.Free;
  Counter := 0;
end;
```

A SaveAs1Click metódus a régi ciklust használja minden egyes komponensfolyam mentéséhez, ahogy azt korábban leírtam. Ismét csak figyelmen kívül hagyjuk a művelet során a Panel komponenst. A program a folyam lezárására egy `try-finally` blokkot alkalmaz a hibák kezelésére:

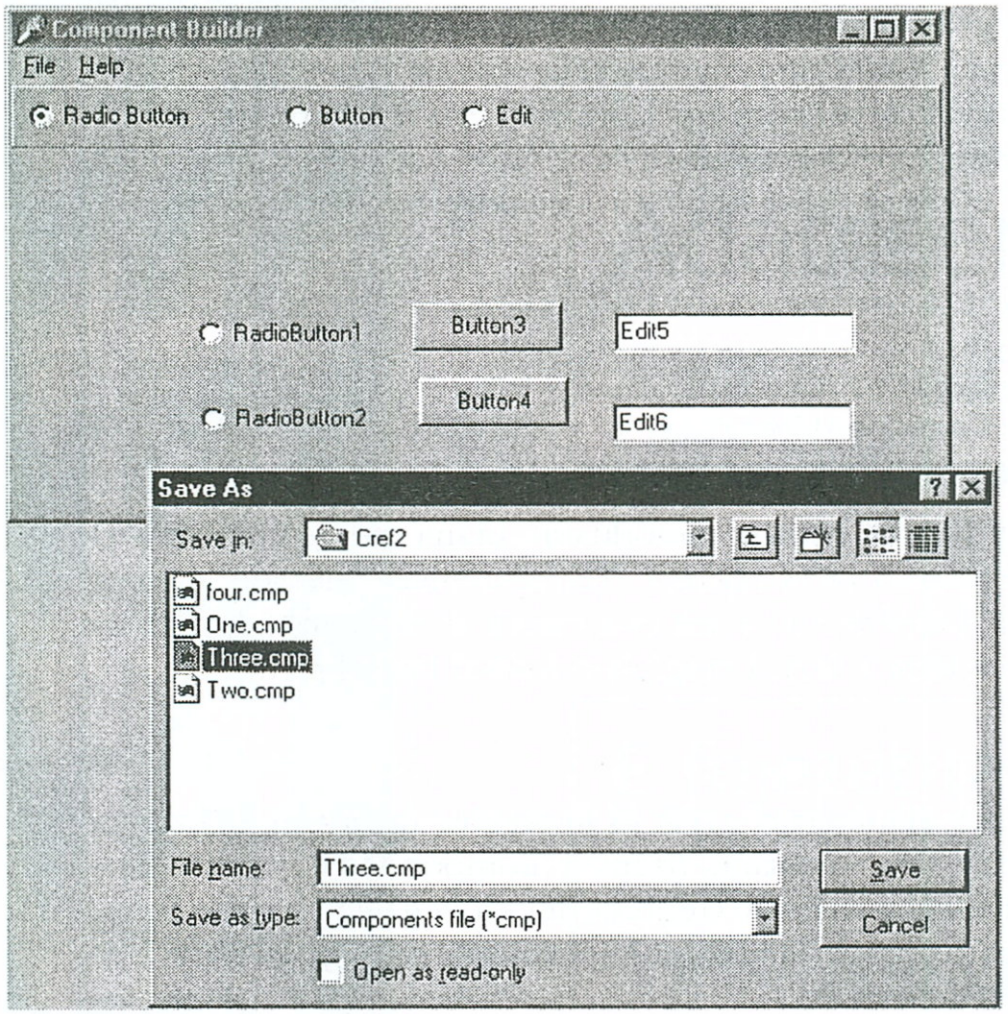
```
procedure TForm1.SaveAs1Click(Sender: TObject);
var
  S: TFileStream;
  I: Integer;
begin
  if SaveDialog1.Execute then
  begin {open or create the stream file}
    S := TFileStream.Create (SaveDialog1.FileName,
      fmOpenWrite or fmCreate);
```

```
      try
        {save each component except the panel}
        for I := 0 to ControlCount - 1 do
          if Controls[I].ClassName <> 'TPanel' then
            S.WriteComponent (Controls[I]);
      finally
        S.Free;
      end;
    end;
  end;
```

A 29.8. ábrán a Save as párbeszédablak megjelenítését láthatjuk a SaveAs1Click metódus által. Az Open1Click metódus nem sokban különbözik ettől. Ebben az esetben a program ciklusa addig fut, amíg végére nem ér a folyamnak. A ciklus a ReadComponent metódust hívja meg, átadva neki egy `nil` paramétert, amely azt jelenti, hogy új komponenst kell létrehozni. Ahelyett, hogy új komponenst hoznánk létre, hozzárendelhetjük a folyamról olvasott értéket egy már létező komponenshez (amelynek ugyanolyan a típusa):

```
procedure TForm1.Open1Click(Sender: TObject);
var
  S: TFileStream;
  New: TComponent;
begin
  if OpenDialog1.Execute then
  begin
    {remove existing controls}
    New1Click (self);
    {open the stream}
    S := TFileStream.Create (OpenDialog1.FileName,
      fmOpenRead);
    try
      while S.Position < S.Size do
      begin
        {read a component and add it to the form}
        New := S.ReadComponent (nil);
        InsertControl (New as TControl);
        Inc (Counter);
      end;
    finally
      S.Free;
    end;
  end;
end;
```

Minden egyes alkalommal, amikor egy új komponens töltődik be, az hozzáadódik a form Controls listájához, az InsertControl metódust használva (ennek természetesen szüksége van egy TControl típusú paraméterre). Ezután a számláló növekszik egyel, elkerülve ezzel a későbbi zavarokat duplikált nevek esetén.



29.8. ábra *A CRef2 példában fájlba menthetjük a komponenseket, de beolvasni is csak ezzel a programmal lehet majd őket*

Az olvasási műveletek működéséhez az egység inicializációs részébe beépítettem egy RegisterClasses rendszereljárást, amely létrehozza az osztálynevek és osztályhivatkozások listáját a folyamot olvasva. Ez azt jelenti, hogy a program csak olyan fájlokat képes olvasni, amely olyan komponenstípusokat tartalmaz, amelyek ebben a függvényben vannak regisztrálva. Ha a programot új komponensekkel akarjuk kiegészíteni, akkor hozzáadhatjuk a megfelelő osztályokat a listához:

```
initialization
  RegisterClasses ([TRadioButton, TEdit, TButton]);
end.
```

Ha egy komponens adatait elmentettük, már nem sokat tudunk vele csinálni. Jobb megoldásnak tűnik, hogy az egész formot lementjük a Delphi DFM formátumában. Más szavakkal: az egész formot el szeretnénk menteni minden komponensével egy Windows erőforrásfájlba, a Delphi által használt standard formátumban. Igazság szerint a DFM fájlok RES fájlok, saját erőforrásokkal.

A kód, ami ezt megvalósítja, sokkal egyszerűbb annál, amit láttunk. Egyszerűen meg tudjuk hívni a WriteComponentResFile eljárást, amelynek egy fájlnévre van szüksége paraméterként (nem egy folyamra) és a form nevére, amit el akarunk menteni. A Cref2 példaprogram File | Save Form File menüpontjához tartozó metódus kódja a következő (a második SaveDialog komponenst használja):

```
procedure TForm1.SaveAs2Click(Sender: TObject);
begin
  if SaveDialog2.Execute then
    WriteComponentResFile (SaveDialog2.Filename, self);
end;
```

Ha elmentünk egy formot ezzel a metódussal, utána megnyithatjuk a Delphi szerkesztőjében egyszerű DFM fájlként. Természetesen minden komponenst meg fogunk találni, beleértve a panelt és a menüt is, de nem tudjuk használni, hacsak nem hozzuk létre a hozzá megfelelő fejlécű Pascal forrásfájlt.

Hogyan tovább?

A fájlok kezelése sok alkalmazás egyik fő funkciója, így a Delphi számos eszközt biztosít ennek támogatására. Számos komponens tartalmaz speciális metódusokat tartalma elmentésére (rendszerint szöveg vagy bitmap formátumban) egy fájlba, és léteznek fájlokhoz kapcsolódó osztályok, mint a Stream osztályok és a TIniFile osztály.

Továbbra is használhatjuk a hagyományos Pascal fájlkezelést, amely a nyelv kulcseleme, valamint a Delphi egy fontos tulajdonsága, hogy minden egyes komponensosztály perzisztens – el tudja menteni saját magát egy folyamra vagy fájlba. Ez számos érdekes Delphi alkalmazás készítéséhez vezet (ennek taglalása azonban már nem ennek a könyvnek a célja).

A fájlokat tipikusan arra használjuk, hogy adatokat mentsünk el későbbi használatra, de arra is, hogy adatok mozgását biztosítsuk különböző alkalmazások között, amelyek ugyanazt a fájlformátumot írják, illetve olvassák. Bár létezik még számos lehetőség erre a Windowsban, beleértve a Vágólapot és a DDE-t is. Ez a két dolog a következő fejezet témája lesz. Egy másik igen hatékony technikát, az OLE Automationt pedig a 22. fejezetben már megbeszéltük.

30. FEJEZET

Adatcsere

E fejezet tartalma:

- Mi is a Vágólap?
- A Delphi vágólap-támogatása
- Dynamic Data Exchange – technikai áttekintés
- DDE a Delphiben
- DDE csatolások másolása és beillesztése
- A DDE használata időzítők és grafikák esetében

Adatcsere

A Windows felhasználók gyakran futtatnak több alkalmazást egy időben. Általában az alkalmazásoknak saját ablakuk van, és látszólag ezek az ablakok függetlenek egymástól. Ám technikailag szerencsére ez egyáltalán nincs így. Valójában az egyes alkalmazások kapcsolatban állnak egymással, így aztán lehetőség nyílik közöttük adatátvitelre.

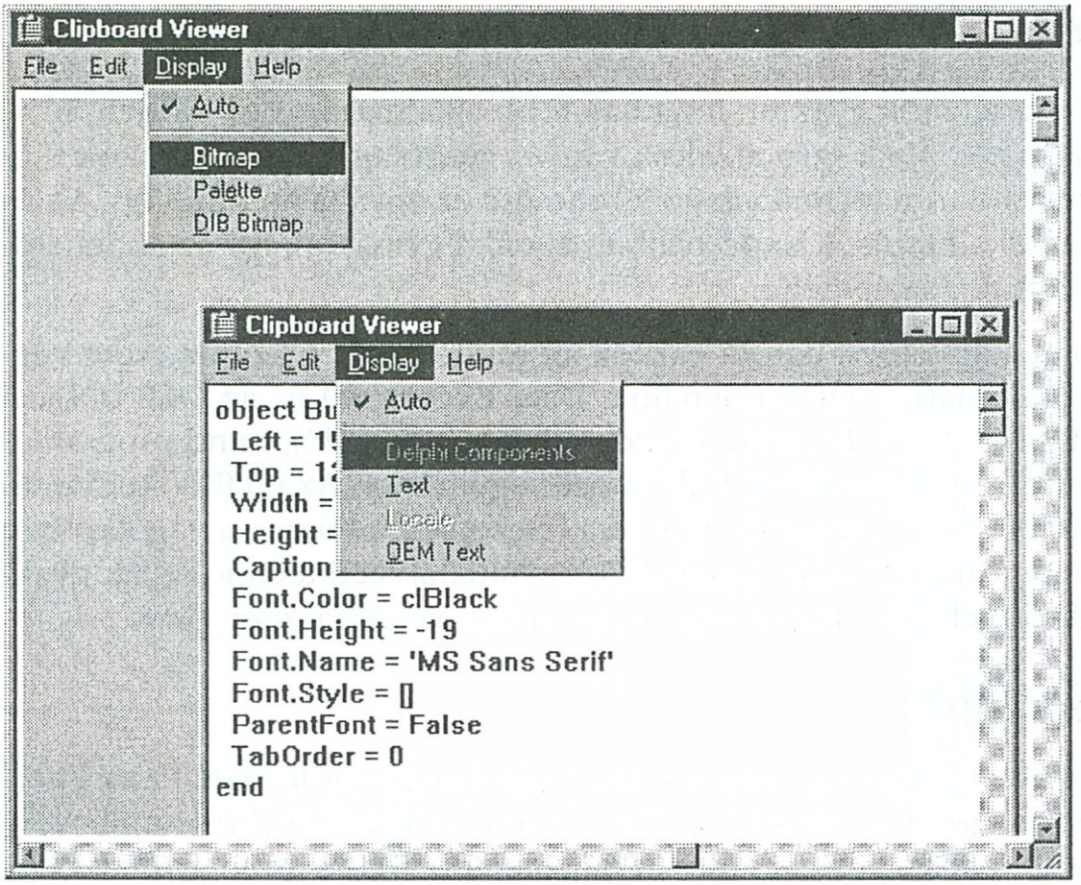
A Windowsban alapvetően három eszköz áll rendelkezésre az adatcsere végrehajtására. A vágólap, a DDE (Dynamic Data Exchange) és az OLE (Object Linking and Embedding). E három eszköz mind felhasználói, mind programozói szempontból sok tekintetben különbözik egymástól. Az OLE dokumentumokat a 22. fejezetben tárgyaltuk. Ennek a fejezetnek a témája a vágólap és a DDE használata. Amint látni is fogjuk, ezek az adatátviteli technológiák gyakran együttműködnek, és sokszor nem is könnyű különválasztani őket.

Mi is a vágólap?

Alapjában véve a vágólap egy információtárolásra alkalmas terület. Ez az információ lehet szövegrészlet, bittérkép, OLE objektum, speciális formátumú adat, stb. Mivel minden adattípushoz tartozik egy megfelelő formátum, ezért ha egy alkalmazás adatot másol a vágólapra akkor a formátumot is meg kell adnia. Ez lehet a szabványos formátumok egyike, de az alkalmazás definiálhat új vágólap formátumot is. Sajnos a helyzet még ennél is bonyolultabb. Előfordulhat, hogy a vágólapon lévő információ egy időben több különböző formátumban is elérhető, hiszen bármelyik alkalmazás megteheti, hogy valamely adatot bemásol a vágólapra több példányban, speciális és szabványos formátumokban is. Ha az adott adatot felhasználó alkalmazás tudja kezelni a speciális formátumot, akkor használhatja azt is. Ha viszont nem tudja, akkor kénytelen a szabványos formátumú adatot felhasználni, ami gyakran információvesztéssel jár.

Vegyük például azt az esetet, amikor kedvenc szövegszerkesztőnkből akarunk adatot juttatni a Windows Jegyzettömbbe. A szövegszerkesztő a betűtípusokra és a szövegformázásra vonatkozó információkat is bemásol a szöveggel a vágólapra. Viszont a Jegyzettömb nem tudja kezelni mindezt az információt, így csak azt a sima szövegfájlt fogja beolvasni, amit a szövegszerkesztő mintegy másodlagosan másolt be a vágólapra.

A vágólap tartalmát a Clipboard Viewer alkalmazás segítségével tekinthetjük meg. Ez az alkalmazás az egyik szabványos formátumban mutatja meg az egyes adatokat, valamint lista formájában megadja, hogy még milyen formátumokban érhetőek el. Mindezt a 30.1. ábra illusztrálja. Ha most ismerkedünk még csak a vágólap programozásával, akkor javaslom a Clipboard Viewer tanulmányozását, mert segítségével közelebb kerülhetünk a vágólap működésének megértéséhez.



30.1. ábra *A Windows 95 vágólap-böngésző az aktuális adat elérhető formátumaira vonatkozó listával. Ezúttal a vágólap egy olyan bittérképet tartalmaz, ami a képernyő egy részletét tartalmazza, amiben pedig maga a vágólap-böngésző van egy Delphi komponenssel a belsejében*

A vágólap „felhasználó-vezérelt" adatátviteli lehetőséget kínál. A felhasználónak kell kimásolnia (illetve kivágnia) az adatot a forrásalkalmazásból, majd átváltani a célalkalmazás ablakába (ha ez utóbbi egyáltalán különbözik a forrásalkalmazástól), s ezek után beilleszteni az adatot. Az egész művelet a felhasználó ellenőrzése alatt történik. Ennek a módszernek az a hátránya, hogy ha az eredeti adat megváltozik, akkor a felhasználónak meg kell ismételnie az egész műveletsort, ahhoz hogy a változások érvényre jussanak a célalkalmazásban is. A DDE és az OLE részben azért készült, hogy ezt a hiányosságot pótolja. Ez a két technológia két teljesen különböző módszerrel dolgozik, azonban mindkettő direkt módon tud adatot összekapcsolni a forrásalkalmazással.

A másik fontos szolgáltatása a vágólapnak az, hogy támogatja az „egyszeres másolás, többszörös beillesztés" műveletét. Ez azt jelenti, hogy ha új adatot másolunk a vágólapra, akkor a régi tartalom elveszik ugyan, de a beillesztés művelete nem semmisít meg semmit, azaz ha beillesztünk egy adatot egy alkalmazásba, attól az még elérhető marad a vágólapon, így akárhányszor felhasználhatjuk.

Amint azt bizonyára mindenki tudja, a vágólappal kapcsolatos parancsok az alkalmazások Edit (szerkesztés) menüjében találhatóak. Ezeket a parancsokat a következő nevekkel illetjük: Kivágás (Cut), Másolás (Copy) és Beillesztés (Paste). A hozzájuk tartozó gyorsbillentyűk pedig azonos sorrendben CTRL-X, CTRL-C és CTRL-V. Az Szerkesztés (Edit) menü gyakran tartalmaz más egyéb vágólap parancsokat is. Ilyenek lehetnek a Csatolás (Paste Link), és az Irányított beillesztés (Paste Special). Amint a következőkben látni fogjuk, ez utóbbiak tulajdonképpen DDE illetve OLE parancsok.

A Delphi vágólap-támogatása

A Delphi két különböző módon támogatja a vágólap használatát:

- Bizonyos komponensek speciális vágólap-kezelő metódusokkal rendelkeznek. Például többek között a `TMemo`, a `TEdit` és a `TDBImage` komponensek tartalmazzák a CopyToClipboard, a CutToClipboard, és a PasteFromClipboard metódusokat.
- A `TClipboard` osztálynak van egy globális Clipboard objektuma, amely számos speciális vágólap-műveletet tartalmaz. Ha teljes vágólap-támogatást akarunk, akkor a Clipboard egységben lévő Clipboard objektumot kell használnunk.

A Clipboard objektum segítségével a program megvizsgálhatja, hogy egy bizonyos formátumban elérhető-e az adat (HasFormat metódus), lekérdezheti az összes elérhető formátum listáját, valamint behelyezhet adatot a vágólapra (már amennyiben ezt nem valamelyik másik komponens intézi). A Clipboard objektum arra is használható, hogy több különböző formátumban másoljunk át egy adott adatot. Ez az egyetlen olyan eset, amikor meg kell nyitni, és be kell zárni a vágólapot a Delphiben. Ez akkor is kötelező lenne, ha közvetlenül Windows API-t használnánk.

Szöveg másolása és beillesztése

Az első vágólapos példánk a 28. fejezetből ismert ClipNote nevű PrnNotes program új változata lesz. Nem lesz nehéz ezt a programot vágólap támogatással bővíteni. Egyszerűen csak engedélyezni kell a Cut, a Copy, és a Paste parancsokat, megfelelően beállítani gyorsbillentyűiket, majd pedig megírni az alábbi három egyszerű metódust:

```
procedure TNotesForm.Copy1Click(Sender: TObject);
begin
  Memo1.CopyToClipboard;
end;
```

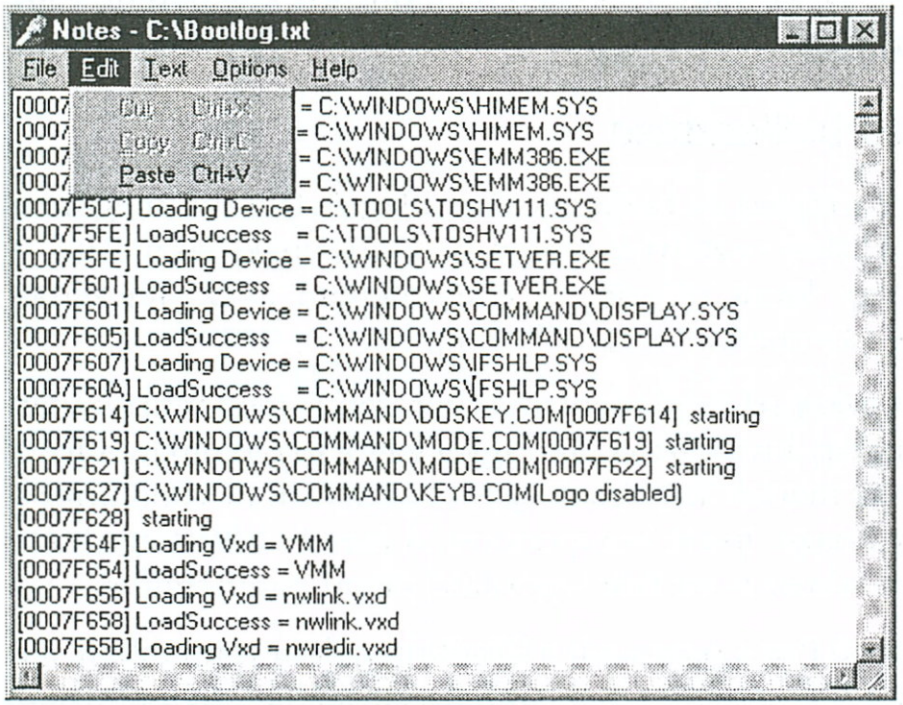
```
procedure TNotesForm.Cut1Click(Sender: TObject);
begin
  Memo1.CutToClipboard;
end;
procedure TNotesForm.Paste1Click(Sender: TObject);
begin
  Memo1.PasteFromClipboard;
end;
```

Most már futtathatjuk a programot, és használhatjuk a vágólapot. Vegyük észre, hogy az első két parancs, a Cut és a Paste, a Memo komponensben levő kijelölt szöveggel dolgozik. A Paste felül is tudja bírálni az aktuális kijelölést (ha volt ilyen), valamint be tudja szúrni a szöveget az aktuális pozícióra. A program működik, ám ez a három parancs nem mindig végzi el a nekik szánt feladatot. Például amikor nincs kijelölés és nincs mit másolni illetve kivágni, vagy amikor a vágólap nem tartalmaz szövegformátumú adatot és nincs mit beilleszteni. A megoldás természetesen az, hogy csak akkor engedélyezzük a megfelelő menüpontokat, miután az aktuális kijelölést és a vágólap tartalmát leellenőriztük.

De hova tegyük ezt a kódot? Nem tehetjük a menüpontok eseménykezelőjébe, mert a programvégrehajtás során az már túl későn lenne. Nem tehetjük a Memo komponenssel foglalkozó részbe sem, mert nem tudhatjuk, hogy esetleg egy másik alkalmazás nem írta-e át a vágólap tartalmát. Legyen hát az Edit menü OnClick eseménykezelőjében. Ha hozzávesszük az Edit1Click metódust a formhoz, akkor a program megfelelően fog működni:

```
procedure TNotesForm.Edit1Click(Sender: TObject);
begin
  {if some text is selected in the memo,
  enable the cut and copy commands}
  if Memo1.SelLength > 0 then
  begin
    Copy1.Enabled := True;
    Cut1.Enabled := True;
  end
  else
  begin
    Copy1.Enabled := False;
    Cut1.Enabled := False;
  end;
  {if the Clipboard contains some text,
  enable the Paste command}
  if Clipboard.HasFormat (CF_TEXT) then
    Paste1.Enabled := True
  else
    Paste1.Enabled := False;
end;
```

Mielőtt lefordítanánk ezt a programot, ne felejtsük el a `uses` kulcsszónál megadni a Clipbrd egységet. Ennek a kódnak a hatását le is tesztelhetjük, ha futtatjuk a programot. A 30.2. ábrán láthatjuk, hogy hogyan viselkedik a program, amikor nincs kijelölt szöveg, de a vágólap tartalmaz szövegformátumú adatot.



30.2. ábra *A ClipNote program, amikor nincs kijelölt szöveg, de a vágólap tartalmaz szövegformátumú adatot*

A fenti listában láthatunk példát a globális Clipboard objektum használatára. A Delphi Súgója öt különböző formátumról tesz említést a HasFormat metódus kapcsán: `CF_TEXT`, `CF_PICTURE`, `CF_BITMAP`, `CF_OBJECT` és `CF_METAFILE`.

Ezeket a formátumokat általában Delphi és VCL komponensek használják, a Windows API azonban egyéb formátumokat is definiál. Ezek közül néhány:

```
CF_BITMAP           CF_DSPMETAFILEPICT
CF_OWNERDISPLAY     CF_SYLK
CF_DIB              CF_DSPTEXT
CF_PALETTE          CF_TEXT
CF_DIF              CF_METAFILEPICT
CF_PENDATA          CF_TIFF
CF_DSPBITMAP        CF_OEMTEXT
CF_RIFF             CF_WAVE
```

Ezeket a Windows formátumokat minden további nélkül használhatjuk, bár a Delphi konkrétan nem támogatja ezeket az adattípusokat. Ebben a példában a `TMemo` osztály két metódusát használtuk a megfelelő vágólap műveletek

elvégzésére, de ugyanezt a hatást elérhettük volna a TClipboard osztály megfelelő szövegmanipulációs eljárásaival is. Például használhattuk volna az AsText tulajdonságot (a stringek másolására, beillesztésére), és a SetTextBuf valamint a GetTextBuf metódusokat (a PChar stringek kezelésére). A TClipboard osztály kifejezetten szöveges műveleteket támogat. Ha más elemekkel akarunk dolgozni, akkor vagy az Assign metódushoz kell nyúlni, vagy pedig kezelőket kell használni.

Bitképek másolása és beillesztése

A Delphiben bitképek másolására és beillesztésére a legtriviálisabb megoldás a TClipboard és a TBitMap osztályok Assign metódusainak használata. A vágólap használatában egy kicsit előrelépve elkészítettem a 22. fejezetből ismert PrintBmp példa új változatát. A neve ClipBmp lett.

Az új példaprogram (a Preview párbeszédablakon keresztül) meg tudja mutatni, és ki is tudja nyomtatni egy kiválasztott fájl vagy a vágólap tartalmát. A form struktúrája mindig ugyanaz marad. Tartalmaz egy – egész formra vonatkozó – TabControl, valamint egy Image típusú komponenst. A menü viszont egy kicsit összetettebb lett, mivel tartalmazza az Edit legördülő menü parancsait is.

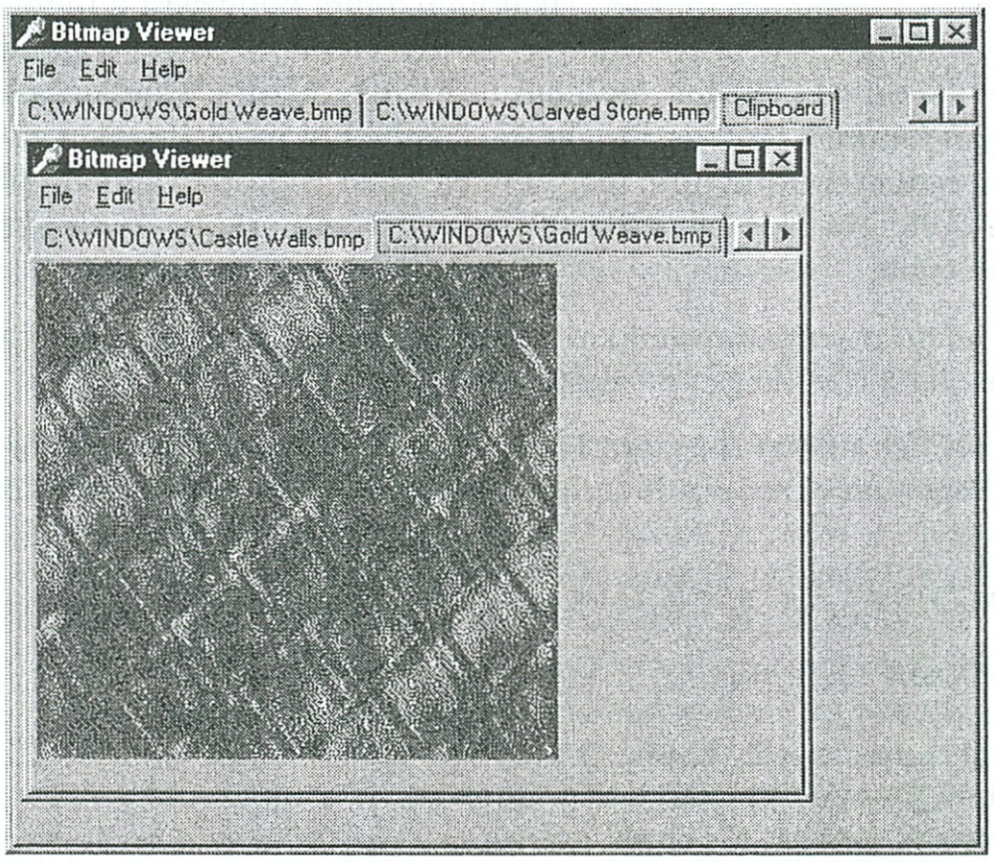
Ha kiválasztjuk az Edit menü Paste parancsát a ClipBmp példaprogramban, akkor az új fül hozzáíródik a fülkészlethez (tab set), ha egyáltalán még nincs ott (lásd 30.3. ábra). Ezek után az új fül sorszámán keresztül lehet megváltoztatni az aktív fület:

```
procedure TForm1.Paste1Click(Sender: TObject);
var
  TabNum: Integer;
begin
  {try to locate the page}
  TabNum := TabControl1.Tabs.IndexOf ('Clipboard');
  if TabNum < 0 then
      {create a new page for the Clipboard}
      TabNum := TabControl1.Tabs.Add ('Clipboard');
  {go to the Clipboard page and force repaint}
  TabControl1.TabIndex := TabNum;
  TabControl1Change (self);
end;
```

A Paste1Click metódus végén a program meghívja a TabControl1Change-et. Ez az eseménykezelő új fül kiválasztásakor aktivizálódik, és bittérképet tud betölteni az aktuális fájlból, illetve beilleszteni a vágólapról:

```
procedure TForm1.TabControl1Change(Sender: TObject);
var
  TabText: string;
begin
```

```
    Image1.Visible := True;
    TabText := TabControl1.Tabs [TabControl1.TabIndex];
    if TabText <> 'Clipboard' then
      {load the file indicated in the tab}
      Image1.Picture.LoadFromFile (TabText)
    else if Clipboard.HasFormat (cf_Bitmap) then
    begin
      {if the tab is 'Clipboard' and a bitmap
      is available in the Clipboard}
      if Image1.Picture.Graphic = nil then
        Image1.Picture.Graphic := TBitmap.Create;
      Image1.Picture.Graphic.Assign (Clipboard);
    end
    else
    begin
      {else remove the Clipboard tab}
      TabControl1.Tabs.Delete (TabControl1.TabIndex);
      if TabControl1.Tabs.Count = 0 then
        Image1.Visible := False;
    end;
  end;
```



30.3. ábra *A ClipBmp példa Clipboard lapja a vágólap aktuális tartalmát mutatja meg amennyiben az egy bitmap. Vegyük észre, hogy hasonlóan a 30.1. ábrához, már megint képet tettem képbe*

Figyeljük meg, hogy ha az Image komponens Picture tulajdonsága nincs inicializálva, akkor az Assign metódus meghívása előtt létre kell hozni a bittérképet. Ha elfelejtjük létrehozni az új bitképet, és a képhez nincs hozzárendelve semmilyen grafika, akkor az Assign művelet sikertelen lesz (és kivételt generál). Ez azért van, mert az Assign metódus nem konstruktor, hanem egy objektum metódusa. S ha az objektumot előzőleg nem hozzuk létre, akkor valamely metódusának meghívásakor GPF hiba keletkezik.

Megjegyzés

Az Assign metódus nem készít másolatot a bittérképről, csak lehetővé teszi, hogy két TBitmap objektum ugyanarra a memóriaterületre mutasson, és ugyanazzal a bitmap kezelővel (handle) rendelkezzen.

Vegyük észre, hogy ez a program minden alkalommal újra beolvassa a bittérképet a vágólapról valahányszor fület váltunk. Ez azt jelenti, hogy a program egyszerre csak egy képet tud tárolni, így nem tudja megőrizni a vágólapbeli bittérképet. Azonban (amint a fenti listából kitűnik) mihelyst megváltozik a vágólap tartalma, és a bittérkép formátum már nem elérhető, akkor a vágólap fül automatikusan törlődik. Ha már egyetlen fül sem elérhető, akkor az Image komponens rejtetté válik.

Képet a Cut és a Delete menüparancsokkal lehet törölni. A Cut megszünteti a fület miután bemásolta a bittérképet a vágólapra.

Gyakorlatilag a Cut1Click metódus a Copy1Click és Delet1Click metódusok hívásán kívül semmi egyebet nem tesz. A Delete egyszerűen eltünteti az aktuális fület.

```
procedure TForm1.Delete1Click(Sender: TObject);
begin
  with TabControl1 do
  begin
    if TabIndex >= 0 then
      Tabs.Delete (TabIndex);
    if Tabs.Count = 0 then
      Image1.Visible := False;
  end;
end;
```

A Copy1Click metódus dolga az aktuális kép bemásolása a vágólapra:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign (Image1.Picture.Graphic);
end;
```

A program többi része a korábbi verziókhoz hasonlóan fájlokat nyit meg, és hozzárakja őket a fülekhez, a File1Click és az Edit1Click metódusokban pedig engedélyezi, illetve letiltja az egyes menüpontokat:

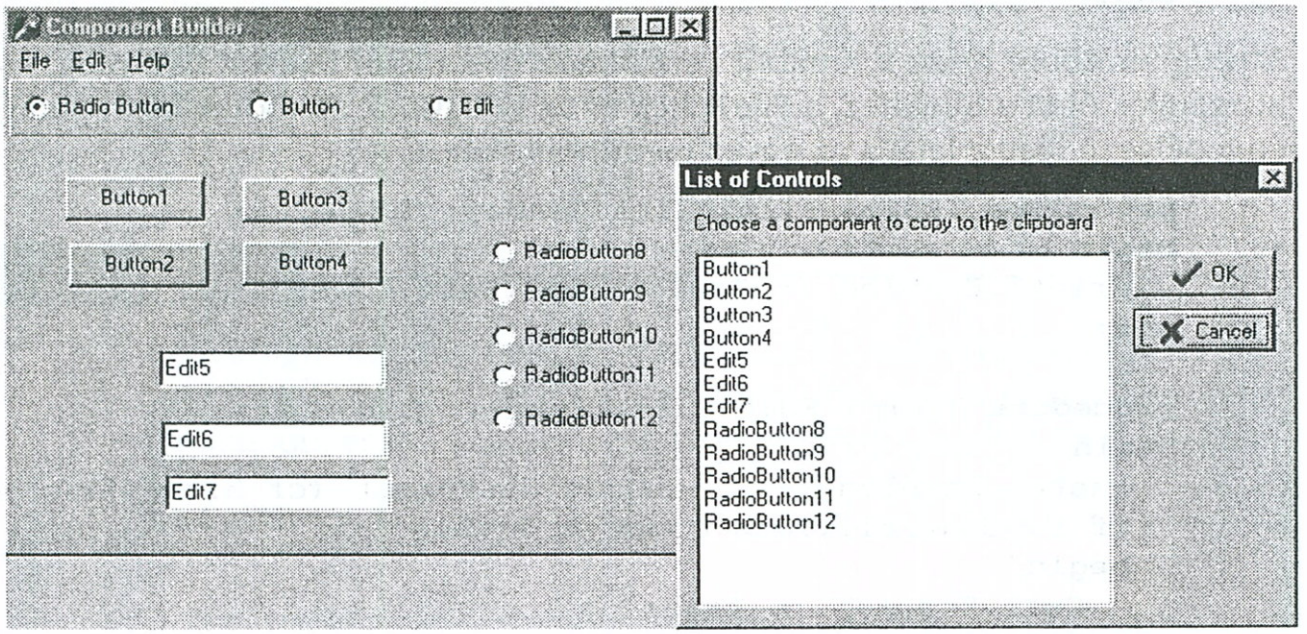
```
procedure TForm1.File1Click(Sender: TObject);
begin
  Print1.Enabled := TabControl1.Tabs.Count > 0;
end;

procedure TForm1.Edit1Click(Sender: TObject);
begin
  Paste1.Enabled := Clipboard.HasFormat (cf_Bitmap);
  if TabControl1.Tabs.Count > 0 then
  begin
    Cut1.Enabled := True;
    Copy1.Enabled := True;
    Delete1.Enabled := True;
  end
  else
  begin
    Cut1.Enabled := False;
    Copy1.Enabled := False;
    Delete1.Enabled := False;
  end;
end;
```

Delphi komponensek bemásolása a vágólapra

Nem csak szöveget és bittérképeket másolhatunk a vágólapra, hanem tetszőleges – akár speciális formátumú – adatot is. Tulajdonképpen ezt csinálja a Delphi is, amikor komponenseket másol. Amikor a Delphi komponenseket másol a vágólapra, akkor az igazi definícióját (bináris kódját), és a szöveges jellemzését is bemásolja. Ez utóbbi bármely szövegszerkesztővel megtekinthető (lásd 2. fejezet). A komponensmásolásra illetve beillesztésre vonatkozó parancsok a `TClipboard` osztály GetComponent, és SetComponent metódusai. Először azt hittem, hogy ezek a metódusok jól jöhetnek Delphi bővítmények írásakor, de amint a későbbiekben látni fogjuk, nem igazán használhatók.

Ezekkel a metódusokkal kiterjeszthetjük az előző fejezetben bemutatott CRef2 példaprogramot (az új program neve CRef3). A program formja ugyanaz marad, csak éppen hozzáveszünk még egy legördülő menüt, amelyben a Copy és

a Paste parancsok kapnak helyet. Ha a felhasználó kiadja a Copy parancsot, akkor a program a 30.4. ábrának megfelelően listát ad a választható kontrolokról.



30.4. ábra *A CRef3 példában a Paste parancs megnyit egy olyan párbeszédablakot, amely az elérhető komponenseket tartalmazza*

A párbeszédablak egy új formban (ListForm) tartalmaz egy alapértelmezett listakeretet, egy címkét, és két szabványos bitmap gombot. Ennek a formnak a manipulálásához szükséges összes kód a fő form Copy1Click metódusában van.

```
procedure TForm1.Copy1Click(Sender: TObject);
var
  I, Index: Integer;
begin  {fill the list of the dialog box}
  ListForm.Listbox1.Clear;
  for I := 0 to ControlCount - 1 do
    if Controls[I].ClassName <> 'TPanel' then
      ListForm.Listbox1.Items.Add (Controls[I].Name);
  if ListForm.ShowModal = mrOk then
  begin
    Index := ListForm.Listbox1.ItemIndex + 1;
    {copy the component and change its name}
    Clipboard.SetComponent (Controls[Index]);
    Controls[Index].Name :=Controls[Index].Name + 'C';
    Controls[Index].Width :=Controls[Index].Width + 10;
  end;
end;
```

A listakeret ItemIndex tulajdonságát – ami egyébként a kiválasztott elem pozícióját mutatja – a Controls vektor számlálójaként használjuk. Itt az a trükk, hogy hozzáadunk egyet az ItemIndex-hez, ezáltal engedélyezzük a panel jelenlétét. A panel pedig a vektor elsőszámú vezérlője.

Vegyük észre, hogy ha átmásolódott a komponens, akkor a neve megváltozik (egy C-vel bővül), hogy akkor se legyen gond, ha a másolatot ugyanabba a formba illesztjük be. Mindez a 30.5. ábrán látható. Ez persze minden alkalommal megtörténik, így aztán ha ugyanazt a komponenst többször másoljuk, akkor a névhez annyi darab C íródik hozzá. Másként is meg lehetett volna oldani a problémát úgy, hogy a Name tulajdonságot üres stringre állítottuk volna, és értelmes értéket adtunk volna a Caption tulajdonságnak. Ám jobbnak láttam mindkét metódusban megváltoztatni a neveket, és minden komponensnek lehetőség szerint egyedi azonosítót adni.

A Paste1Click metódus egyszerűen csak beilleszti az aktuális komponenst a vágólapra úgy, hogy megváltoztatja a pozícióját, nevét, és szélességét (ez utóbbit azért, hogy el tudja helyezni a meghosszabbított nevet):

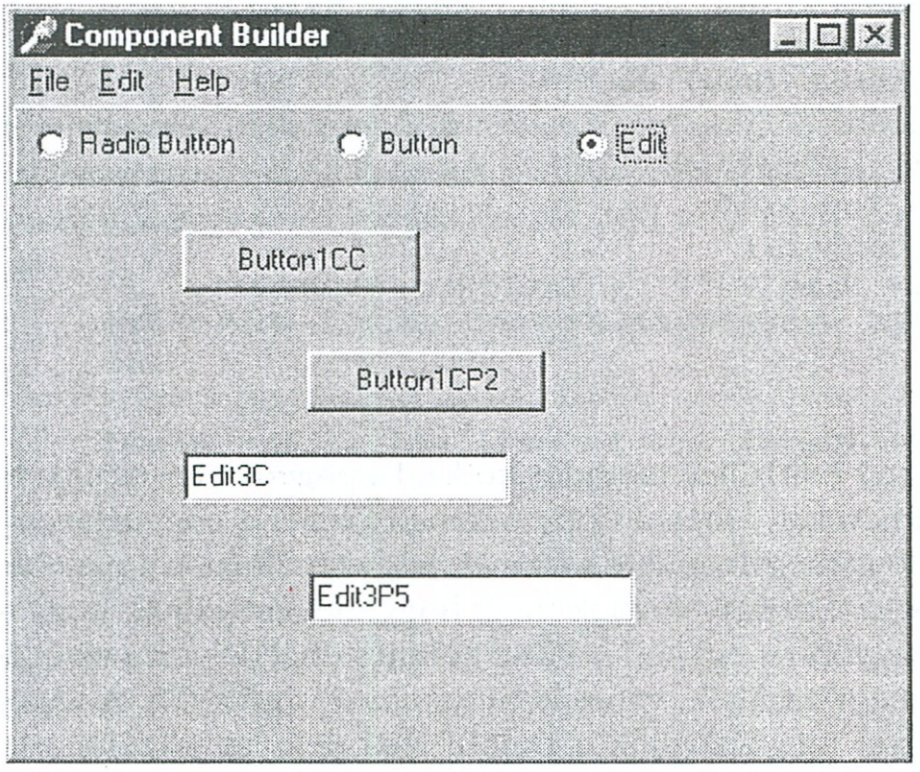
```
procedure TForm1.Paste1Click(Sender: TObject);
var
  New: TComponent;
begin
  {retrieve the component, moving it slightly}
  New := Clipboard.GetComponent (self, self);
  if New is TControl then
    with TControl (New) do
    begin
      Left := Left + 50;
      Top := Top + 50;
      Inc (Counter);
      Name := Name + 'P' + IntToStr (Counter);
      Width := Width + 10;
    end;
end;
```

A legbonyolultabb rész az Edit1Click metódus kódja. Ez engedélyezi és tiltja le a Copy és a Paste parancsokat. Hogy a Copy parancsot lehet-e engedélyezni, úgy határozzuk meg, hogy megvizsgáljuk a formot, hogy tartalmaz-e a panelen kívül más komponenst is. Ezt úgy tehetjük meg, hogy megvizsgáljuk, hogy a ControlCount értéke nagyobb-e egynél. A Paste parancs engedélyezhetőségét úgy döntjük el, hogy megvizsgáljuk, van-e komponens a vágólapon. A vágólapböngésző segítségével megnézhetjük a Delphi által definiált vágólap-formátum nevét, és annak alapján megírhatjuk a kódot. Hogy egy adott komponensformátum rajta van-e a vágólapon, úgy ellenőrizhetjük, hogy meghívjuk a Windows API RegisterClipboardFormat függvényét a keresett formátum nevével paraméterezve. A visszatérési érték egy olyan formátumkód, ami az egész

rendszerben érvényes. Ez egyben azt is jelenti, hogy ha két alkalmazás ugyanazt a formátumnevet használja, akkor szükségszerűen ugyanaz lesz a formátumkódjuk is:

```
procedure TForm1.Edit1Click(Sender: TObject);
var
  Format: Cardinal;
begin
  Format:= RegisterClipboardFormat('Delphi Component');
  Paste1.Enabled:= Clipboard.HasFormat (Format);
  Copy1.Enabled:= ControlCount > 1;
end;
```

A CRef3 program sok irányba bővíthető. Például ha ugyanazt a komponenst kétszer másoljuk át, akkor a régebbi objektum felülíródik. A fő probléma mégis az, hogy a Clipboard metódus által használt formátum nem egyezik meg a Delphi belső formszerkesztője által használt formátummal. A Delphi formszerkesztője a komponens szöveges jellemzését is bemásolja a vágólapra, ám egy másik formátumot is használ. Ennek neve Delphi Components. Sajnos ez a formátum nem kompatibilis a `TClipboard` osztály SetComponent metódusa által használt Delphi Component („s" nélkül) formátummal. Így aztán ez a program nem is igazán jó példa Delphi bővítőeszközök készítésére.



30.5. ábra *Ha a CRef3 példában komponenst másolunk, akkor az eredeti objektum neve megváltozik. A beillesztett objektumokat a P különbözteti meg*

Speciális adat másolása a vágólapra

A következő példaprogram vágólap támogatással lát el egy teljesen hétköznapi, ám speciális adattípussal rendelkező programot. Ehhez a `TClipboard` osztály GetAsHandle és SetAsHandle metódusait alkalmazzuk. Ezek a metódusok pedig a Windows API SetClipboardData és GetClipboardData függvényeit használják fel. Ezek az eljárások egy globális Windows memóriablokkhoz rendelt kezelőt (handle) adminisztrálnak. A memóriablokkot a GlobalAlloc függvényen keresztül foglalják le (ahhoz, hogy a vágólap kezelését magasabb szinten elsajátíthassuk meg kell ismerkedni a Windows memóriablokkok használatával. Ez az ismeret majd a DDE-nél is jól fog jönni).

Most az a cél, hogy előkészítsünk egy memóriablokkot a speciális formátumú információ számára. Ehhez először meg kell hívni néhány Windows API függvényt, amelyekkel memóriát foglalunk le (ez a könnyebbik fele), majd pedig fel kell tölteni a memóriablokkot az adattal. Mindezek megvalósítására a legelterjedtebb módszer az, hogy veszünk egy pointer típusú változót, és azon keresztül írunk, illetve olvasunk a memóriaterületről.

Ez a példa a 29. fejezetből ismert Graph3 példa új változata. Ebben a verzióban új Edit menünk van. Az új menü tartalmazza a Copy és a Paste parancsokat, valamint a szokásos három függvényt, ami a Copy-hoz, Paste-hez, és magának az Edit menünek kiválasztásához tartozik. Máskülönben az alkalmazás formja teljes mértékben megegyezik az előző verzióéval. A program legérdekesebb része valószínűleg a Copy1Click metódus.

Mint már említettem a Windows-ban a GlobalAlloc függvénnyel lehet memóriablokkot lefoglalni. Ez a függvény két paramétert vár: egy flag típusút, ami a memória típusára utal, valamint egy másikat, ami a blokk méretét adja meg. A visszatérési érték egy kezelő (handle), ami a memóriablokkra mutat. Ezt a kezelőt aztán átadhatjuk a GlobalLock nevű API függvénynek, az pedig majd visszaad egy pointert, ami a memóriablokk első pozíciójára mutat:

```
HMem := GlobalAlloc (ghnd, 25 * sizeof (Integer));
PInt := GlobalLock (HMem);
```

Tudjuk, hogy Delphiben ezt a két lépést átugorjuk, és egyszerűen létrehozunk egy objektumot. Sajnos azonban most nem használhatjuk a Delphi memóriakezelését, mert a vágólaphoz szabványos memóriablokk kell. Amint ezt a blokkot átadjuk a vágólapnak, attól kezdve a vágólap lesz a tulajdonosa, és nincs többé jogunk ahhoz, hogy használjuk. Ha van egy érvényes PInt mutatónk, akkor már tudjuk kezelni a memóriablokkot és feltölthetjük értékekkel. Ez történik, amikor egészekre mutató pointereket használunk, azaz úgy használjuk a memóriablokkot, mintha egy egészekből álló vektor lenne. A mi esetünkben a következő két utasítást hívjuk meg újra és újra (ciklusban):

```
PInt^ := Value;
Inc (PInt);
```

Az első utasítás eltárol egy értéket a pointer által mutatott memóriacímre, a második pedig átállítja a pointert a következő memóriarekeszre. Ha a memóriablokk kész van, akkor – amint a metódus kódjából kitűnik – elég meghívni a SetAsHandle metódust a Clipboard objektumból:

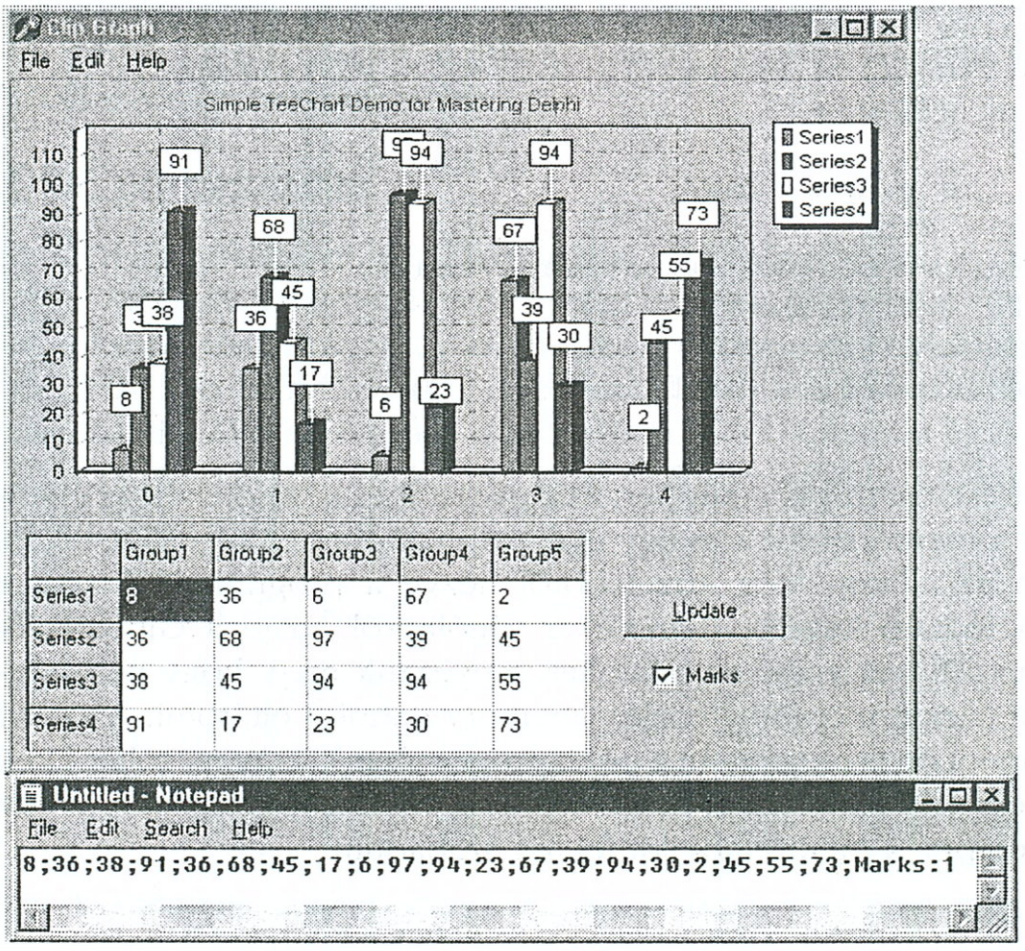
```
procedure TForm1.Copy1Click(Sender: TObject);
var
  ClipForm, HMem: Cardinal;
  Text: String;
  PInt: ^Integer;
  I, J, Value: Integer;
begin
  {register a custom Clipboard format}
  ClipForm := RegisterClipboardFormat ('Chart_Data');
  {allocate a memory block and retrieve
  a pointer to it}
  HMem := GlobalAlloc (ghnd, 25 * Sizeof (Integer));
  PInt := GlobalLock (HMem);
  Text := '';
  {build the data, in both versions}
  for I := 1 to 5 do
    for J := 1 to 4 do
    begin
      {add the number for each cell of the grid}
      Value := StrToIntDef (
        Trim(StringGrid1.Cells [I, J]), 0);
      PInt^ := Value;
      Inc (PInt);
      AppendStr (Text,
        StringGrid1.Cells [I, J] + ';');
    end;
  {add the status of the check box}
  Value := Integer (ChBoxMarks.Checked);
  PInt^ := Value;
  AppendStr (Text, 'Marks:' + IntToStr (Value));
  {open the Clipboard to copy
  multiple versions of the data}
  Clipboard.Open;
  try
    Clipboard.SetAsHandle (ClipForm, HMem);
    Clipboard.AsText := Text;
  finally
    Clipboard.Close;
  end
end;
```

A fenti kódnak van egy másik tulajdonsága is. A program két különböző formában is bemásolja az aktuális információt a vágólapra: egy speciális, és egy

szöveges (string) formátumban. Ahhoz, hogy egy adatot két példányban másolhassunk fel a vágólapra, a TClipboard osztály Open és Close metódusaival ellenőrzés alatt kell tartanunk a vágólapot amíg végrehajtjuk a műveleteket (ellenkező esetben a második adatformátum felülírná az elsőt). Viszont ha megnyitjuk a vágólapot, akkor egy adatot több formátumban is bemásolhatunk (lásd 30.6. ábrán a Clipboard Viewer ablakot).

Vegyük észre továbbá, hogy speciális vágólap-formátumot deklaráltam (Chart_Data). Erre a formátumra az Edit1Click metódusban is szükségem lesz, hogy ellenőrizhessem megvan-e a megfelelő típusú információ a vágólapon:

```
procedure TForm1.Edit1Click(Sender: TObject);
var
  ClipForm: Cardinal;
begin
  {if the proper format is available, enable Paste}
  ClipForm := RegisterClipboardFormat ('Chart_Data');
  Paste1.Enabled := Clipboard.HasFormat (ClipForm);
end;
```



30.6. ábra *A ClipGrph adatot be lehet illeszteni szövegszerkesztőbe is, mivel a program két különböző formátumban is bemásolta az adatot a vágólapra*

A programot fejleszthettük volna olyan irányba is, hogy elfogadjon szöveges bemenő adatot is (a vágólapról). Viszont végigvizsgálni a stringet, hogy az adat érvényes formátumban van-e, majd pedig kibogarászni belőle a megfelelő információt túlontúl bonyolult lenne. Emiatt aztán csak „Chart_Data" típusú memóriablokkot lehet visszailleszteni. Ez egyébként a Paste1Click metódus kódjából is kitűnik:

```
procedure TForm1.Paste1Click(Sender: TObject);
var
  HMem, ClipForm: Cardinal;
  PInt: ^Integer;
  I, J, Value: Integer;
begin
  {get the memory block from the Clipboard}
  ClipForm := RegisterClipboardFormat ('Chart_Data');
  HMem := Clipboard.GetAsHandle (ClipForm);
  PInt := GlobalLock (HMem);
  {read each number}
  for I := 1 to 5 do
    for J := 1 to 4 do
    begin
      Value := PInt^;
      StringGrid1.Cells [I, J] := IntToStr(Value);
      Inc (PInt);
    end;
  {read the graph type}
  Value := PInt^;
  ChBoxMarks.Checked := Boolean(Value);
  {update everything}
  ChBoxMarksClick (self);
  UpdateButtonClick (self);
end;
```

A Paste parancs kezelője tulajdonképpen a Copy parancs kezelőjének az ellentéte. Végigmegy a memóriablokkon egy egész típusú pointer segítségével, és kiolvassa onnan az adatot. Még közelebbről nézve a program kiolvassa a memóriablokkra mutató kezelőt (handle) a vágólapról, majd a GlobalLock függvényen keresztül beolvassa az adatokat. Jegyezzük meg, hogy nem szabad a vágólaptól kapott memóriakezelőt (memory handle) eltárolni. Azonnal ki kell olvasni (amint én is megtettem a példaprogramban), vagy másolatot kell készíteni a róla egy új memóriablokkban. Ugyanis amikor legközelebb adatot másolunk a vágólapra, a memóriablokk fel fog szabadulni.

Dynamic Data Exchange – technikai áttekintés

Amint a fejezet elején tisztáztuk, a vágólap alapjában véve felhasználó-vezérelt. Ha viszont azt akarjuk, hogy egy alkalmazás felhasználói beavatkozás nélkül tudjon adatot átvinni egy másikba, akkor ajánlatos dinamikus adatátvitelt használni. A DDE segítségével kapcsolatot lehet létesíteni két alkalmazás között, a létrejött kapcsolaton keresztül pedig adatot lehet átvinni egyik alkalmazásból a másikba. A DDE sokféle műveletet tud. Például tud kérésre adatot küldeni, vagy változó adatokat folyamatosan frissíteni.

A DDE első változata az MS Excel munkafüzetek (spreadsheet) egy továbbfejlesztése volt. Annak idején ez egy üzenetalapú protokoll volt, ami annyit jelent, hogy az egyes alkalmazások speciális Windows üzeneteket küldözgettek egymásnak, s így történt az információcsere. Ez a protokoll alapjában véve egy szabálygyűjtemény volt, mellyel más programozók által írt alkalmazásokkal lehetett kommunikálni.

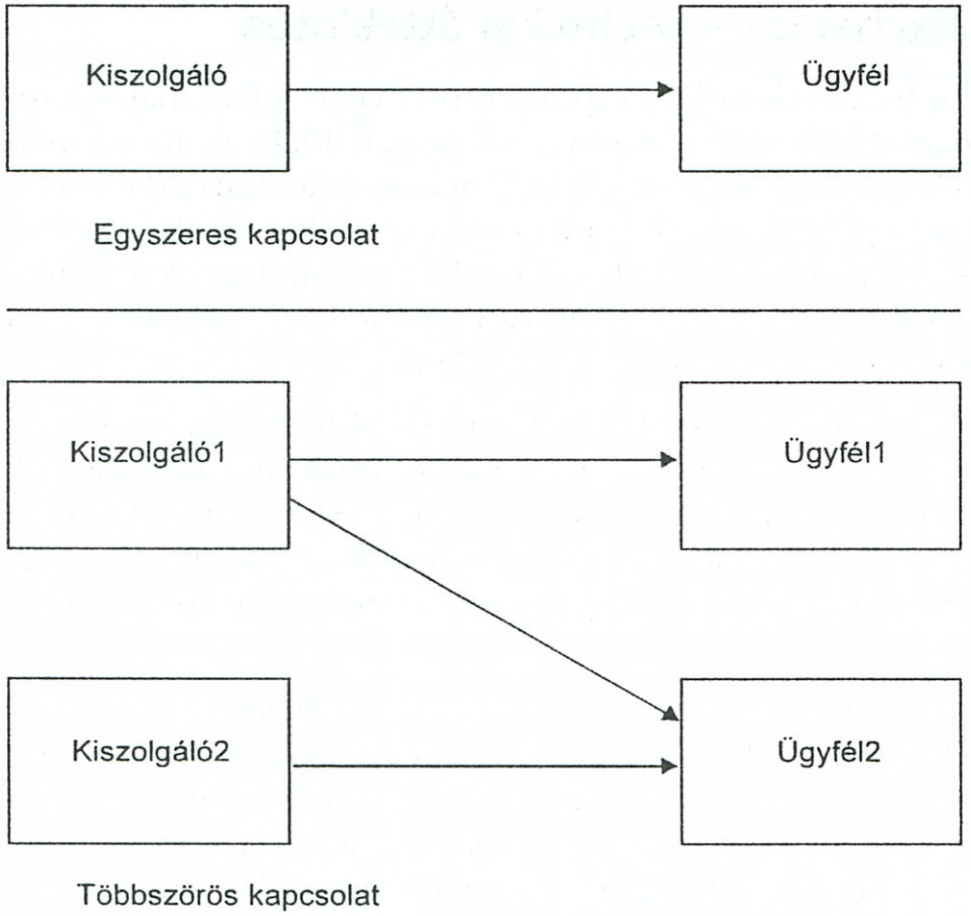
Ahogy telt múlt az idő, a DDE egyre inkább a Windows részévé kezdett válni. A második verziója (DDEML = DDE Management Library) bekerült a Windows 3.1-be. A DDEML információkkal rendelkezik az aktuális DDE eseményekről. Az alkalmazások nem közvetlenül kommunikálnak, hanem a DDEML-en keresztül. Így a DDEML-re hárul a szóbajövő problémák megoldása. Ettől lesz a DDE sokkalta megbízhatóbb.

DDE párbeszédek

A DDE segítségével két alkalmazás között párbeszéd (conversation) jöhet létre. Az egyik alkalmazás szerverként, a másik kliensként viselkedik. A szerver az információszolgáltató, a kliens pedig a folyamat vezérlője. Ha egy alkalmazás DDE kompatibilis, akkor szerverként ki tud szolgálni több klienst is, kliensként pedig több szerverhez is tud kapcsolódni, valamint egyidőben tud kliensként és szerverként működni. A 30.7. ábrán látható néhány DDE párbeszéd grafikus szemléltetése.

A szerver és a kliens szerepe

Milyen szerepet játszik a szerver és a kliens? Mit történik egy DDE párbeszéd során? A szerver feladata természetesen az, hogy kérésre adatot küldjön a kliensnek. A kliens feladata pedig egyrészt a párbeszéd kezdeményezése, másrészt az adat elkérése a szervertől, adatküldés a szerverhez (poke), illetve a szerver felszólítása parancsok végrehajtására (execute) (régebben előszeretettel használták a DDE execute parancsokat, de mára már inkább az OLE Automation került előtérbe ezen a területen).



30.7. ábra *Néhány példa DDE párbeszédekre*

A legfontosabb feladat természetesen az adatküldés. Ha egyszer elkezdődött egy párbeszéd, akkor a kliens adatot kérhet a szervertől (request), illetve elindíthat egy úgynevezett tanácsoló ciklust (advise loop). Tanácsoló ciklus indítása annyit jelent, hogy a kliens megkéri a szervert, hogy egy adott adattal történő változásokról értesítse őt, a szerver pedig minden alkalommal, amikor az adat megváltozik, küld a kliensnek vagy egy jelzést a változásról, vagy egyszerűen egy új példányt az adatrészből. A DDE három elemet használ annak meghatározására, hogy melyik szerverhez lehet kapcsolódni, illetve, hogy mi lesz az adatcsere tárgya:

- A Service tulajdonképpen a DDE szerver alkalmazás neve. Ez lehet a futtatható fájl neve is (kiterjesztés nélkül), de lehet a szerver által meghatározott név is.
- A Topic a párbeszéd globális témája. Ez lehet adatfájl, egy szerver ablaka, vagy bármi más. Egy adott DDE párbeszéd egy szerver és egy kliens között egy adott témával kapcsolatban jön létre.
- Az Item egy adott adatelem azonosítója. Ez lehet egy adatbázis egy mezője, egy táblázat egyik cellája, vagy akár egy részvény a tőzsdén. Egy párbeszéden és témán belül a kliens és a szerver több adatot is cserélhet.

DDE a Delphiben

Az általános bevezető után most már visszakanyarodhatunk a Delphire, illetve annak DDE támogatására. A komponenspaletta System lapja négy DDE-vel kapcsolatos komponenst tartalmaz:

- A DdeServerConv a párbeszéd jellemzését kezeli (beleértve a téma nevét is) a szerver oldalról nézve, továbbá támogatja az execute parancsokat.
- A DdeClientConv a kliens oldalról kezeli a párbeszédet. Ez adja meg az aktuális szervert, és témát, valamint ezt kell használni a párbeszéd megkezdéséhez.
- A DdeServerItem egy párbeszéden belül mutat egy adatelemre a szerveren. Általában ezt az adatot küldjük a klienshez, illetve poke művelet esetén ezt kapjuk a klienstől. Ennek a komponensnek a neve határozza meg a DDE elemet.
- A DdeClientItem a szervertől kapott adatra mutat (illetve poke művelete esetében a szervernek küldött adatra). A szerver DDE elemére mutat.

Egy egyszerű DDE példa

DDE szerver és kliens készítése mindössze annyiból áll, hogy elhelyezünk ezen elemek közül néhányat két különböző alkalmazás két különböző formjában, majd írunk egy néhány soros kódot. Természetesen a DDE célja az, hogy megoldja az alkalmazások közti kommunikációt. Mivel nem tudom, hogy az olvasó számítógépén milyen DDE-s programok vannak, ezért a most következő példákban többnyire mi írjuk mind a szerver, mind pedig a kliens alkalmazást, de fogok mutatni olyan példát is, amelyben más klienst, illetve szervert használunk.

Az első programunk vélhetően a lehető legegyszerűbb szerver, s hamarosan össze is kapcsoljuk majd a lehető legegyszerűbb klienssel. Ha az is megvolt, akkor elkezdjük bonyolítani a dolgot.

Az egyszerű DDE szerver

Az első szerver (FirstSer) formja egy szerkesztőmezőt, és egy DdeServerItem komponenst tartalmaz. Nincs benne DdeServerConv komponens, mert az csak akkor kellene, ha a speciális névvel akarnánk ellátni a párbeszédet, illetve ha execute parancsokat akarnánk kezelni. A DdeServerConv komponens hiányában a szolgáltatás neve az alkalmazás neve (az .EXE kiterjesztést elhagyva), a téma neve pedig a DdeServerItemet tartalmazó form címe lesz.

Ennek a programnak a kódja rendkívül egyszerű. Valahányszor megváltozik az Edit komponens, a program bemásolja a szerkesztőmező tartalmát a DdeServerItem Text tulajdonságába:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  DdeServerItem1.Text := Edit1.Text;
end;
```

Ezt a szerver alkalmazást akár le is futtathatjuk, de sokkal érdekesebb lesz, ha megépítjük hozzá a klienst is.

Az egyszerű DDE kliens

Az első kliensprogram csak egy kicsit bonyolultabb az első szerverprogramnál. Ennek a példaprogramnak a formja tartalmaz egy DdeClientConv, egy DdeClientItem komponenst, valamint egy szerkesztőmezőt. A DdeClientItem1 komponens a DdeConv tulajdonságon keresztül csatlakozik a párbeszédkomponenshez, ám a kirajzolás során nincs kapcsolat a DdeClientConv1 és a szerver között. A kapcsolat csak akkor jön létre, amikor a form már készen van:

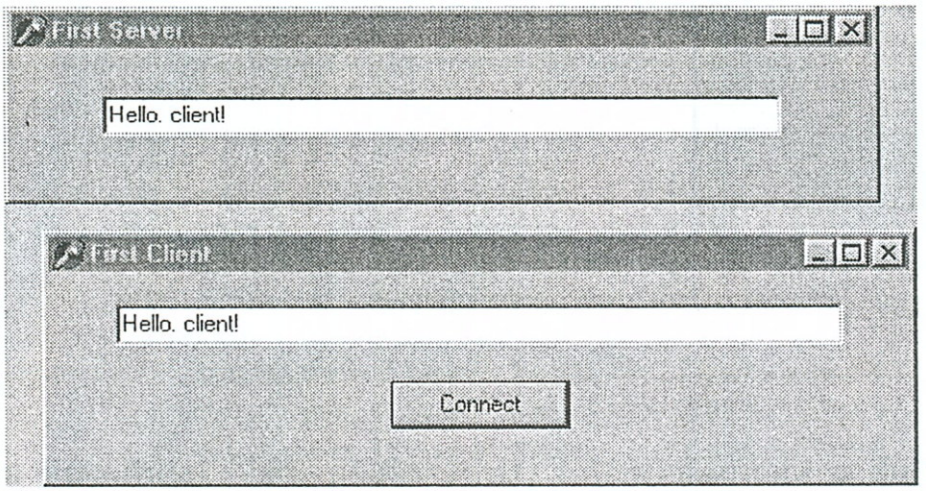
```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if DdeClientConv1.SetLink('firstser', 'First Server')
  then
    begin
      ShowMessage ('Connected');
      DdeClientItem1.DdeItem := 'DdeServerItem1';
    end
  else
      ShowMessage ('Error');
end;
```

A program meghívja a SetLink metódust a szolgáltatással (a szerver nevével) és a témával (a form címével) paraméterezve. Ha a kapcsolat sikeres, akkor egyrészt megjelenik egy üzenet, másrészt az alkalmazás összekapcsolja a kliens-elem DdeItem tulajdonságát a szerverelemmel. Amint létrejött a kapcsolat, és a csatolás aktívvá vált, akkortól kezdve bármikor megváltozik a szerveren lévő adat, a szerver azonnal küldeni kezdi a megváltozott adatot a kliensnek. A kliens alkalmazás egyszerűen bemásolja a saját szerkesztőmezőjébe a kapott adatot:

```
procedure TForm1.DdeClientItem1Change(Sender: TObject);
begin
  Edit1.Text := DdeClientItem1.Text;
end;
```

Ha létrejött a kapcsolat és a csatolás, akkor minden amit a felhasználó begépel a szerver szerkesztőmezőjébe, az azonnal átmásolódik a kliens szerkesztőmezőjébe. A 30.8. ábrán láthatjuk, hogy hogyan jut érvényre a szerver szerkesztőmezőjében létrejött változás a kliensében (természetesen ajánlott a programokat saját kezűleg lefuttatni, hogy teljes mértékben kiélvezhessük az általuk nyújtott lehetőségeket).

Vegyük észre, hogy a DDE csatolás a szerver alkalmazás DdeServerItem komponense, és a kliens alkalmazás DdeClientItem komponense között helyezkedik el. Valahányszor megváltozik a szerverelem Text tulajdonsága, ez a változás rögtön tükröződik a klienselem Text tulajdonságában. Mivel a DDE elemek Text tulajdonságaiban lévő szövegeket, és a két Edit komponens ide tartozó tulajdonságait kapcsoltuk össze, tulajdonképpen két szerkesztőmező között jött létre a kapcsolat.



30.8. ábra *Ha létrejött a kapcsolat, akkor bármit is gépelünk a szerver szerkesztőmezőjébe, az azonnal megjelenik a kliensében is*

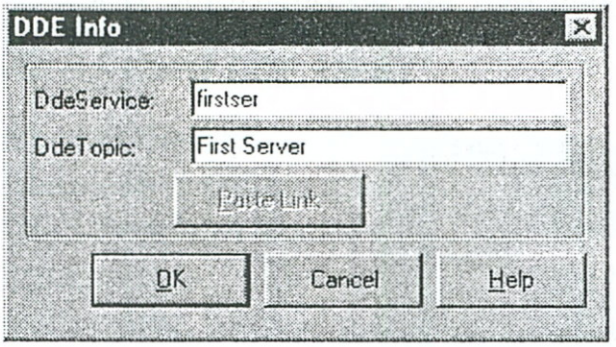
Megjegyzés

Amit ezekben a DDE alkalmazásokban építettünk, az más néven egy DDE tanácsoló ciklus (advise loop, de úgy is emlegetik, hogy hot link). Mindezt a DDEML könyvtár segítségével tehettük meg. Ha ugyanezt a kódot C-ben akarnánk megírni, az bizony eltartana egy darabig, mert a DDEML nem éppen a legegyszerűbb Windows API. Mivel azonban a Delphi támogatja az advise loopokat, ezért ez a program egyszerűbb lett az előzőnél, ahol a kommunikáció a kliens kéréseire alapult. Az a program egyébként a DdeClientConv komponens RequestData metódusán alapul.

DDE kapcsolat tervezési időben

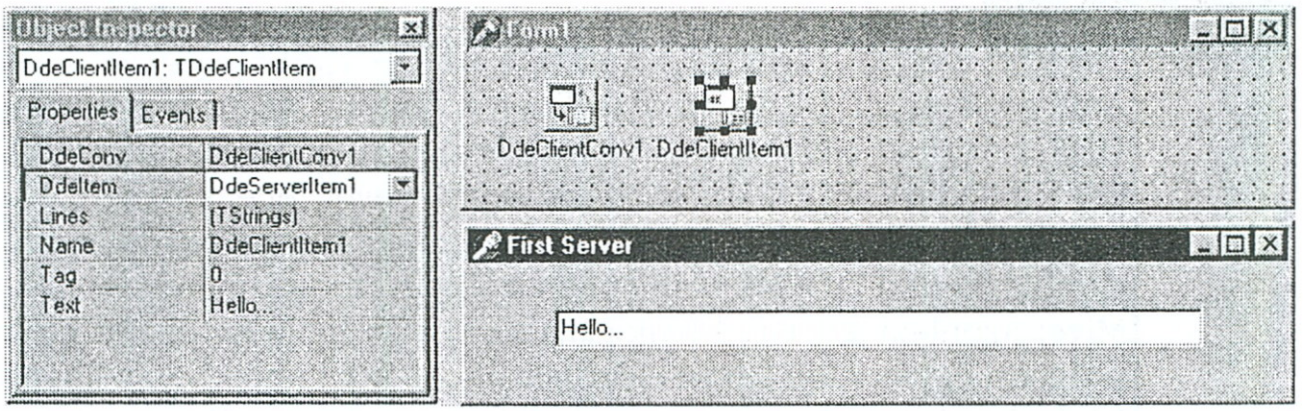
Az előző példában futásidőben hoztuk létre a kapcsolatot. A Delphivel azonban azt is megtehetjük, hogy tervezési időben hozzuk létre a kapcsolatot. Ezt ki is próbálhatjuk, ha futtatjuk a szerver alkalmazást, megnyitjuk a kliens

projektet, kiválasztjuk a DdeClientConv1 komponenst, és megnyitjuk a DdeService vagy a DdeTopic tulajdonság tulajdonságszerkesztőjét. A megjelenő párbeszédablakba a 30.9. ábrán látható módon írjuk be a kívánt információt.



30.9. ábra *A DdeClientConv komponens DdeService és DdeTopic tulajdonságainak tulajdonságszerkesztői*

Most már kiválaszthatjuk a klienselemet, a párbeszédet, majd beírhatjuk a DdeItem tulajdonság nevét, s lám a kapcsolat létrejött. Most változtassunk meg valamit a szerveren, és az új szöveg már látható is az Object Inspectorban (30.10. ábra).



30.10. ábra *Ha fut a szerver (jobb alsó sarok), és létrejött a kapcsolat, akkor a kliens Text tulajdonsága automatikusan frissítődik a szerveren lévő szöveg alapján*

Ha kirajzolási időben hozzuk létre a kapcsolatot, akkor a Connect gombot, és a kódját akár el is hagyhatjuk. Elég futtatni a kliens programot, és a DDE kapcsolat automatikusan létrejön (persze ha fut a szerver is). A magam részéről jobbnak tartom a futásidejű kapcsolatfelvételt, mert az nem követeli meg, hogy a két alkalmazást meghatározott sorrendben indítsa el a felhasználó.

DDE csatolások másolása és beillesztése

A következő lépés az lesz, hogy a programot a másolás és a beillesztés támogatás DDE-s változatával bővítjük. Ezt úgy visszük véghez, hogy egy kicsit változtatunk a kódon, és megengedjük a programnak, hogy kapcsolatba lépjen egy harmadik alkalmazással.

Kapcsolt adat másolása a vágólapra

Hasonlóan a korábbiakhoz az első lépés ennél a DDE példánál is a szerver megépítése lesz. Az új szerverprogram (PasteSer) formja három szerkesztőmezőt, három DdeServerItem komponenst, és három gombot tartalmaz. Ez a verzió a DdeServerConv komponenst is felhasználja. Ennek a programnak a kódja is még elég egyszerű. Amikor a form létrejön futásidőben, akkor a szerkesztőmezőkben lévő szöveg bemásolódik a megfelelő szerverelembe:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DdeServerItem1.Text := Edit1.Text;
  DdeServerItem2.Text := Edit2.Text;
  DdeServerItem3.Text := Edit3.Text;
end;
```

Ez a szöveg másolódik benne újra és újra, amikor valamelyik szerkesztőmező tartalma megváltozik. Íme a három metódus egyike:

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
  DdeServerItem2.Text := Edit2.Text;
end;
```

Az új kód a gombok OnClick eseménykezelőjében van. Ha a felhasználó rákattint az egyik gombra, akkor a szerkesztőmezőben lévő szöveg, és a csatolás is bemásolódik a vágólapra:

```
procedure TForm1.CopyButton1Click(Sender: TObject);
begin
  Clipboard.Open;
  try
    Clipboard.AsText := DdeServerItem1.Text;
    DdeServerItem1.CopyToClipboard;
  finally
    Clipboard.Close;
  end
end;
```

A szerver másolás művelete a kapocsra vonatkozó információt másolja. Ez az információ nem a tényleges adat. Ha csak a kapcsot másoljuk, akkor az esetleges harmadik alkalmazás nem biztos, hogy fel fogja ismerni az adatot, mert

semmi információt nem bocsátottunk rendelkezésére az adat formátumáról. Ezért nem elég csak a kapcsot bemásolni a vágólapra.

E kóddal kimásolhatjuk valamelyik szerkesztőmező tartalmát, majd bemásolhatjuk szinte bármely DDE kliensbe. Például a Microsoft Word for Windowsba ezt az adatot a Beillesztés, és a Csatolás paranccsal is beilleszthetjük. Indítsuk el a PasteSer programot, másoljuk ki az egyik szöveget, nyissunk meg egy Word dokumentumot, majd válasszuk ki a Szerkesztés menüből az Irányított beillesztés parancsot. A párbeszédablakban válasszuk ki a Csatolás rádiógombot, és kattintsunk az OK-ra. Ezzel létrejött a kapcsolat.

Most a szöveg bekerült a Word dokumentumba, és ha átmegyünk a szerverre és átírjuk a szöveget, akkor az a Word dokumentumban is meg fog változni. Ez bizonyára nem meglepő, hiszen itt ugyanarról van szó, mint az előző példában. Ugyanezt a módszert alkalmazhatjuk más Windows alkalmazásokban is. A megfelelő menüpontok nevei ha nem is ugyanezek, de hasonlóak.

A DDE használata időzítők és grafikák esetében

Delphiben a legtöbb DDE program szöveges adattal kommunikál. Ez azért van így, mert a Delphi a szöveges elemek támogatja a leginkább. Ha viszont a Delphi DDE támogatás használata mellett meghívunk néhány függvényt a DDEML API-ból akkor ez az apró hátrány kiküszöbölhető. Ám általában egyszerűbb konvertálni az adatot szövegformátumból oda-vissza, és aztán kihasználni a Delphi DDE támogatását, mintsem a Delphi dokumentálatlan tulajdonságaival ügyeskedni, meg közvetlen API hívásokat használni.

Készítettem egy példát, ahol grafikát viszünk át, hogy az olvasó lássa, hogy a Delphi nem csak szöveget tud kezelni. Igaz, az adatátvitel ezúttal is stringeken keresztül történik. A most következő példa két szempontból különbözik az előzőtől. Egyrészt a szerveradat automatikusan kerül frissítésre egy időzítő segítségével, másrészt a kliens grafikusan mutatja az adatot.

Az automata szerver

A szerver (DataServ) formja igen egyszerű. Tartalmaz egy csak olvasható Memo komponenst, egy időzítőt, egy Copy gombot, és egy DdeServerItem komponenst. Ha a form létrejött, a program generál egy véletlen számot a Randomize függvénnyel. Mindig, amikor az időintervallum letelik, a Memo soraiban levő adatok megváltoznak, mégpedig olymódon, hogy mindegyikhez hozzáadódik egy -10 és +10 közötti szám (hozzáadódik, illetve kivonódik egy 1 és 10 közötti szám). A kezelő végén a Memo szövege bemásolódik a DdeServerItembe:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  I, Value: Integer;
```

```
begin
  for I := 0 to 4 do
  begin
    Value := StrToIntDef (Memo1.Lines , 50);
    Value := Value + Random (21) - 10;
    Memo1.Lines  := IntToStr (Value);
  end;
  DdeServerItem1.Lines := Memo1.Lines;
end;
```

A másik metódus a vágólapra másolás művelete. Ez minden bizonnyal ismerős lesz. Nem csak az adatot másoljuk át, hanem a DDE csatolást is:

```
procedure TForm1.CopyButtonClick(Sender: TObject);
begin
  Clipboard.Open;
  try
    Clipboard.AsText := Memo1.Text;
    DdeServerItem1.CopyToClipboard;
  finally
    Clipboard.Close;
  end;
end;
```

Ha erről a szerverről másolunk, és csatoljuk az eredményt egy alkalmazásba, akkor bármiféle felhasználói beavatkozás nélkül is mindig automatikusan frissülni fog az adat.

Grafikus DDE kliens

Az, hogy automatikusan változó számokat tudunk megjeleníteni egy harmadik alkalmazásban nagyon izgalmas, ám eddig csak szöveggel dolgoztunk. A most következő kliens alkalmazás grafikusan fogja megjeleníteni az adatot. A ViewData kliens formja csak a két tipikus DDE kliens párbeszédeket, és a DDE klienselem komponenseket tartalmazza, nem nincs benne semmi speciális tulajdonság.

A kapcsolat a form elkészülte után jön létre. Mivel a form ilyen egyszerű, azt várhatjuk, hogy a kód viszont bonyolult. Az is...

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if DdeClientConv1.SetLink ('dataserv',
       'Data DDE Server') then
    DdeClientItem1.DdeItem := 'DdeServerItem1'
  else
    ShowMessage ('Start the server before the client');
end;
```

Megpróbálhatjuk azt is, hogy hibaüzenet helyett inkább elindítjuk a szervert programból, a WinExec API segítségével. Ez viszont bonyolulttá is válhat, ha a szerver nem az aktuális könyvtárban, vagy a PATH környezeti változó valamelyik könyvtárában van.

Valahányszor megváltozik a DDE elem adata, az új értékeket kiolvassuk a stringből, és bemásoljuk egy egészekből álló vektorba. Ezt a vektort a TForm1 osztály privát adataként definiáljuk:

```
Values: array [0..4] of Integer;
```

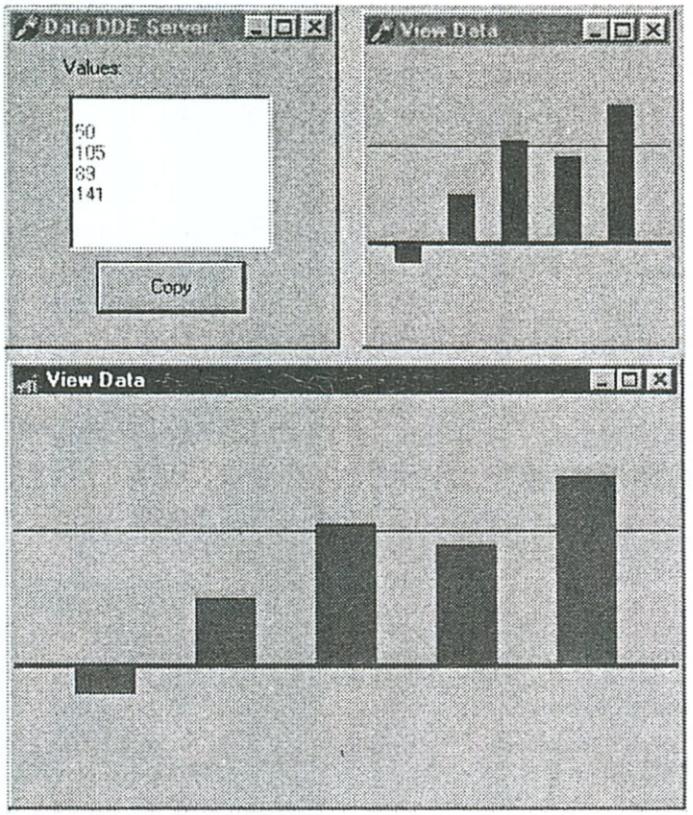
Először azt hittem, hogy a szerverelem sorai másolódtak be a klienselem soraiba, és ennek a kódnak csak az lesz a dolga, hogy kicsomagolja, illetve konvertálja a stringet. Viszont a program végül egyetlen stringet adott vissza, amely tartalmazta az összes számot. Némi kísérletezés után rájöttem a megoldásra. A DdeClientConv komponens FormatChars tulajdonságát True értékre kell állítani, ezzel megakadályozzuk, hogy a program átugorja a soremelés karaktereket. Ha beállítottuk ezt a tulajdonságot, akkor írjuk be a következő kódot:

```
procedure TForm1.DdeClientItem1Change(Sender: TObject);
var
  I: Integer;
begin
  {extract the numbers}
  for I := 0 to 4 do
    if I < DdeClientItem1.Lines.Count then
  Values  := StrToIntDef (DdeClientItem1.Lines, 50);
  Invalidate;
end;
```

Az `if` blokk redundánsnak tűnhet, de nem az, hiszen ez a metódus induláskor hívódik meg, amikor az értékek még nincsenek ott. S most néhány szó a központi kódról. Az DdeClientItem komponens OnChange eseménykezelőjének utolsó sora (fentebb) meghívja az Invalidate-et ami újrarajzolja a formot.

A FormPaint metódus elég bonyolult. Ha tanulmányozni kívánjuk, akkor megtalálhatjuk a CD mellékleten.

Az újrarajzoló kód eredménye a 30.10. ábrán látható. A program két példányban fut, hogy látni lehessen, hogy a kép automatikusan méreteződik a form méretéhez képest. A ViewData példában a sávok felhasználói beavatkozás nélkül automatikusan módosulnak a kliens és a szerver alkalmazásban egyaránt.



30.11. ábra

Két példa a ViewData program futásának eredményére

Hogyan tovább?

Ebben a fejezetben megvizsgáltuk, hogy hogyan kell vágólap és DDE támogatással bővíteni a programjainkat. Megnéztük továbbá a két módszer közti hasonlóságokat, és különbségeket. A vágólap-használat nagyon elterjedt a Windows alkalmazásokban. Szerencsére a vágólap-támogatás megvalósítása általában elég egyszerű (még akkor is, ha használunk API-t). Ezzel ellentétben a DDE alkalmazása már komolyabb odafigyelést igényel. Az, hogy a felhasználó nem csak adatot, hanem csatolásokat is tud másolni és beilleszteni, már önmagában elég izgalmas. Ami még izgalmasabb, az az, hogy felhasználóink más népszerű Windows alkalmazásokkal is cserélhetnek adatot. A Delphiben elég egyszerű a DDE megvalósítása, különösen, ha figyelembe vesszük, hogy milyen bonyolult kód húzódik meg a háttérben. Az egyetlen probléma az, hogy a Delphi DDE támogatása nincs igazán jól dokumentálva, és nem minden DDE szolgáltatás érhető el egykönnyen. Ennek ellenére a Delphi DDE támogatása a legtöbb programozási feladathoz elégségesnek mondható.

Itt az idő, hogy rátérjünk a könyv utolsó témájára, a Delphi multimédiás lehetőségeire. Nem fogjuk teljes mélységében tárgyalni a témát, csupán nagy vonalakban ismertetjük a Delphi által nyújtott kép- és hangkezelési technikákat.

31. FEJEZET

Multimédiás lehetőségek

E fejezet tartalma:

- A Windows alapértelmezett hangjai
- A Médialejátszó komponens
- A CD meghajtó használata

Multimédiás lehetőségek

A fájlrendszer és a nyomtató kezelésének módját már megismerhettük a 28. és a 29. fejezetben, most további hardver egységekkel foglalkozunk: a hangkártyával és a CD-ROM-mal (természetesen, ahhoz, hogy Delphi programjainkban használhassuk ezeket az eszközöket, megfelelően telepítve kell lenniük a Windows alatt).

A Windows egy speciális API-val, a Multimedia API-val támogatja a videó, a MIDI és a CD eszközök használatát. A Delphi is egy különálló komponenssel (Médialejátszó) és egy különálló súgófájllal segíti ezen külső multimédia-eszközök használatát. Bár e komponens használata sem bonyolult, mielőtt ezt megnéznék, ismerkedjünk meg néhány egyszerűbb módszerrel számítógépünk dalra fakasztásához.

A Windows alapértelmezett hangjai

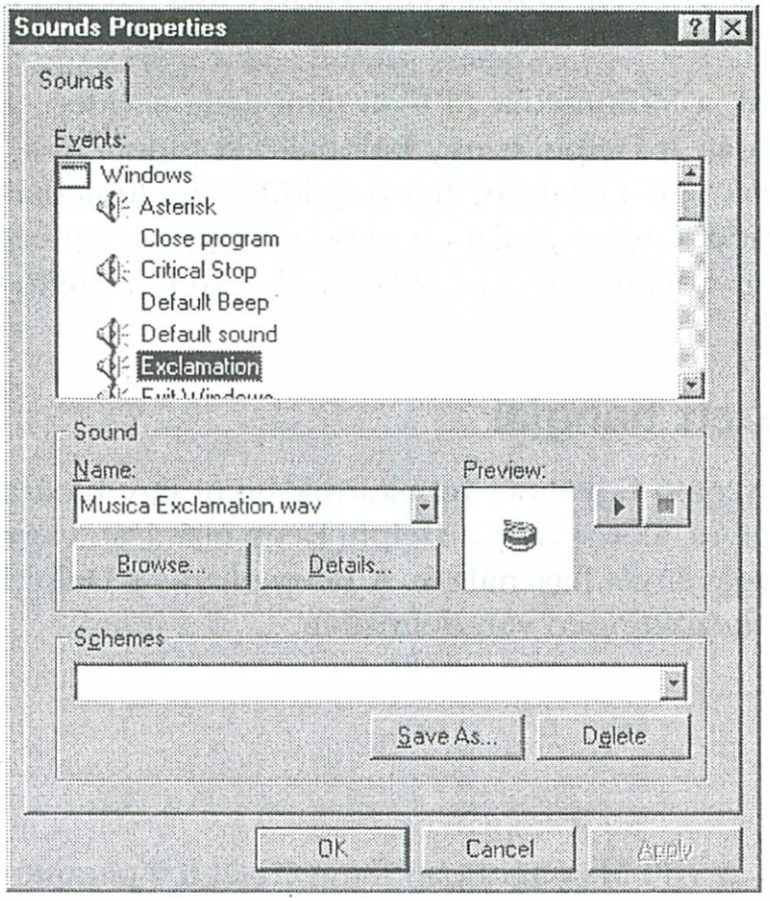
Eddig könyvünkben, ha a felhasználó figyelmét fel akartuk hívni egy hibára vagy valamilyen speciális eseményre, akkor vagy a Delphi Beep eljárását, vagy a Windows API MessageBeep függvényét használtuk. A Beep eljárás a Delphi futásidejű könyvtárában, a következőképpen van definiálva:

```
procedure Beep;
begin
  MessageBeep(0);
end;
```

Tehát egyszerűen csak meghívja az API MessageBeep függvényét a 0 paraméterértékkel. A MessageBeep függvényhívás a 0 és a -1 értékekkel a számítógép belső hangszóróját szólaltatja meg, a további lehetséges értékekkel a Windows alapértelmezett hangjait játszhatjuk le. A paraméterként használható konstansok és a nekik megfelelő Windows hangok (a Vezérlőpult elnevezései szerint) a következők:

mb_IconAsterisk	SystemAsterisk	Üzenetablak
mb_IconExclamation	SystemExclamation	Figyelmeztetés
mb_IconHand	SystemHand	Előugró menü
mb_IconQuestion	SystemQuestion	Kérdés
mb_Ok	SystemDefault	Alapértelmezett hang

Az eseményekhez rendelt hangfájlokat a Vezérlőpulton állíthatjuk be (31.1. ábra), ahol a felsorolt nevekhez rendelt hangokat módosíthatjuk. Ezek a beállítások a Windows rendszerbeállítások (Windows System Registry) között kerülnek elmentésre.



31.1. ábra *Az alapértelmezett Windows hangok beállítása a Vezérlőpulton*

Ugyanezek a konstansok használhatóak az API MessageBox függvényében, amit a `TApplication` osztály MessageBox metódusa hív meg. Ezeket a hangokat általában akkor szólaltatjuk meg, amikor megjelenítjük az üzenetablakot. A Delphi MessageDlg eljárása ezt nem teszi meg önműködően, de könnyedén létrehozhatjuk a SoundMessageDlg eljárást, ami már nemcsak a megfelelő ikont jeleníti meg, hanem hangjelzést is ad. Ezt a következő példában be is mutatjuk majd.

Minden ablaknak megvan a maga hangja

A MessageBeep API-hívás bemutatására a Beeps példaprogramot készítettem. Ennek formján egy rádiócsoportban a fenti konstansoknak megfelelő rádiógombok vannak. A rádiócsoport szöveges definíciója a következő:

```
object RadioGroup1: TRadioGroup
  Caption = 'Parameters'
  ItemIndex = 0
  Items.Strings = (
    'mb_IconAsterisk'
    'mb_IconExclamation'
    'mb_IconHand'
    'mb_IconQuestion'
    'mb_Ok')
end
```

A Beep Sound gomb megnyomásakor a program lejátssza a megfelelő hangfájlt. Ezen gomb OnClick eseményének kezelője először egy `case` utasítás segítségével meghatározza, hogy melyik rádiógombot választottuk ki, majd lejátssza a megfelelő zenét.

```
procedure TForm1.BeepButtonClick(Sender: TObject);
var
  BeepConstant: Cardinal;
begin
  case RadioGroup1.ItemIndex of
    0: BeepConstant := mb_IconAsterisk;
    1: BeepConstant := mb_IconExclamation;
    2: BeepConstant := mb_IconHand;
    3: BeepConstant := mb_IconQuestion;
    4: BeepConstant := mb_Ok;
  else
    BeepConstant := 0;
  end;
  MessageBeep (BeepConstant);
end;
```

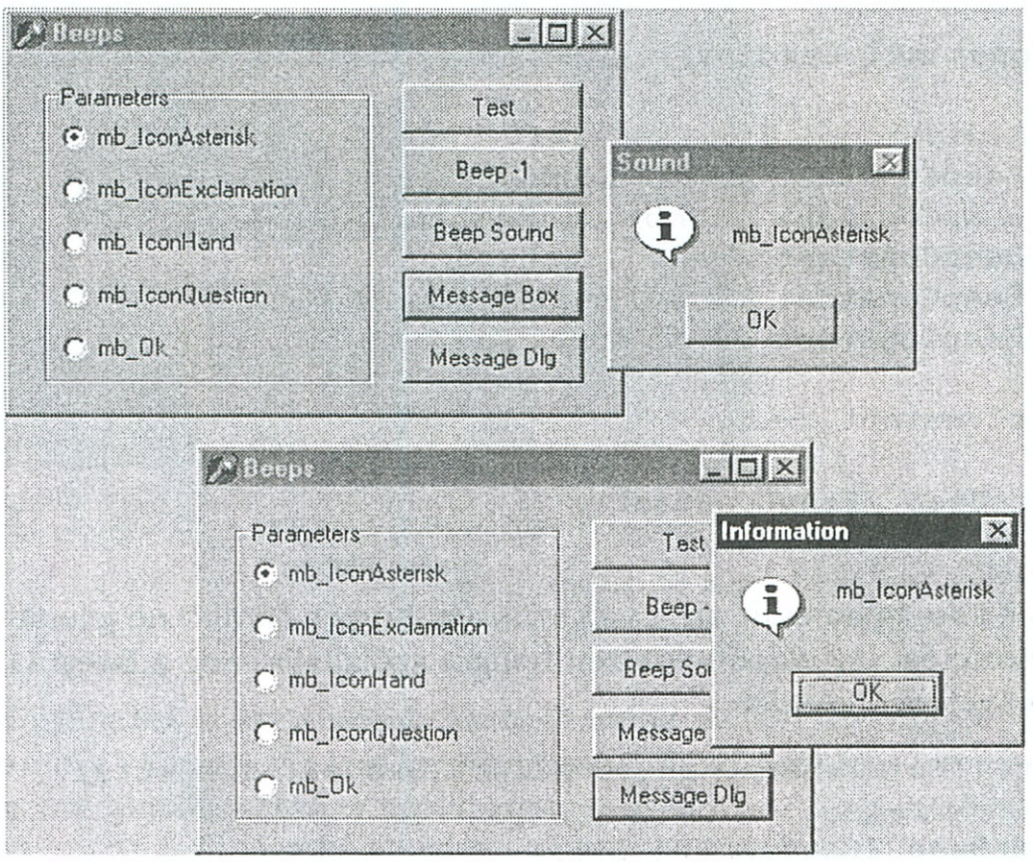
A `case` utasítás `else` ágára csak azért van szükség, hogy a fordító ne jelenítsen meg figyelmeztetést. Az alapértelmezett hangot szólaltatja meg a **Beep -1** nyomógomb a következő kóddal:

```
procedure TForm1.BeepOneButtonClick(Sender: TObject);
begin
  MessageBeep (Cardinal (-1));
end;
```

Ugyanezt az eredményt érhetjük el azzal is, ha a MessageBeepnek a $FFFFFFFF hexadecimális értéket adjuk át. Annak megállapításához, hogy van-e telepített hangkártya-meghajtó a számítógépen (ez nem jelenti, hogy hangkártya is van a gépben, mert a beépített hangszóróhoz is léteznek meghajtóprogramok) a Test nyomógombot használhatjuk, ami egy multimédia függvényt, a WaveOutGetNumDevs-t hívja a teszteléshez:

```
procedure TForm1.TestButtonClick(Sender: TObject);
begin
  if WaveOutGetNumDevs > 0 then
    SoundMessageDlg ('Sound is supported',
      mtInformation, [mbOk], 0)
  else
    SoundMessageDlg ('Sound is NOT supported',
      mtError, [mbOk], 0);
end;
```

Ahhoz, hogy ezt az eljárást le tudjuk fordítani, az MmSystem egységet is fel kell sorolnunk a `uses` részben. Ha nincs hangkártya-meghajtó a számítógépben, akkor csak a szokásos csipogást hallhatjuk, függetlenül attól, hogy melyik hangot választottuk ki. Az utolsó két gomb egy-egy üzenetablakot jelenít meg a megfelelő hangjelzéssel együtt (31.2. ábra).



31.2. ábra *A MessageBox és a SoundMessageDlg függvény kimenete*

A Message Box nyomógomb OnClick eseményének kezelője hagyományos Windows megközelítést használja. Először meghívja a MessageBeep függvényt, majd az Application objektum MessageBox metódusát. Ez azt eredményezi, hogy először a hangfájl lejátszása kezdődik el, s közben jelenik meg az

üzenet. Valójában (a hangkártya típusától függően) a hangkártya vezérlése nem állítja meg a többi Windows működést. A gombhoz rendelt kód ezek alapján a következő:

```
procedure TForm1.BoxButtonClick(Sender: TObject);
var
  BeepConstant: Cardinal;
begin
  case RadioGroup1.ItemIndex of
    0: BeepConstant := mb_IconAsterisk;
    1: BeepConstant := mb_IconExclamation;
    2: BeepConstant := mb_IconHand;
    3: BeepConstant := mb_IconQuestion;
  else {including 4:}
    BeepConstant := mb_Ok;
  end;
  MessageBeep (BeepConstant);
  Application.MessageBox (
    PChar (RadioGroup1.Items [RadioGroup1.ItemIndex]),
    'Sound', BeepConstant);
end;
```

Az utolsó gombra kattintva a program a SoundMessageDlg függvényt hívja meg. Ez nem egy beépített Delphi eljárás, de minden programunkban hasznos lehet. Ha rendszeresen használni akarjuk, érdemes rövidebb nevet választanunk. A SoundMessageDlg az AType paraméterben megadott zenét játssza le, és utána a szokásos üzenetablakot jeleníti meg (31.2. ábra).

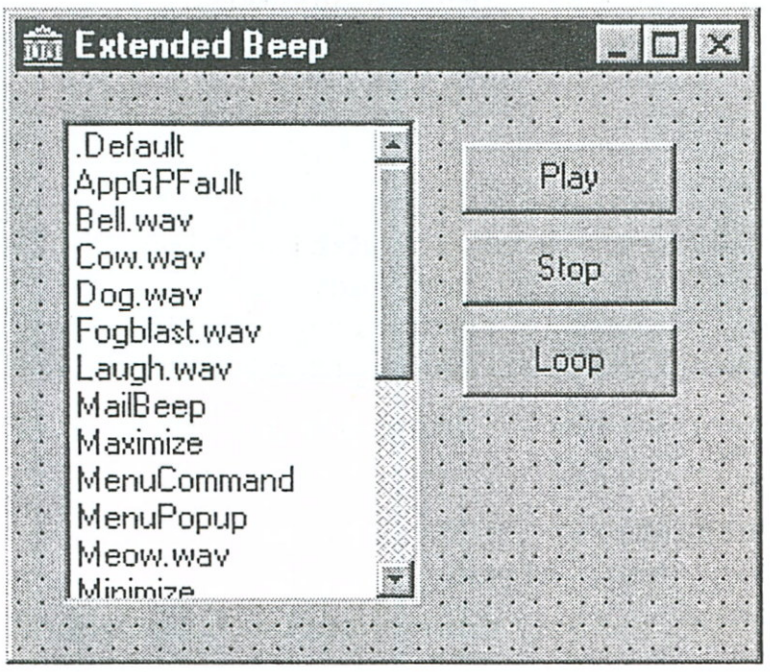
```
function SoundMessageDlg (const Msg: string;
  AType: TMsgDlgType; AButtons: TMsgDlgButtons;
  HelpCtx: Longint): Integer;
var
  BeepConstant: Cardinal;
  begin
  case AType of
    mtWarning: BeepConstant := mb_IconExclamation;
    mtError: BeepConstant := mb_IconHand;
    mtInformation: BeepConstant := mb_IconAsterisk;
    mtConfirmation: BeepConstant := mb_IconQuestion;
  else
    BeepConstant := mb_Ok;
  end;
  MessageBeep(BeepConstant);
  Result := MessageDlg (Msg, AType,
    AButtons, HelpCtx);
end;
```

```
procedure TForm1.MessDlgButtonClick(Sender: TObject);
var
  DlgType: TMsgDlgType;
begin
  case RadioGroup1.ItemIndex of
    0: DlgType := mtInformation;
    1: DlgType := mtWarning;
    2: DlgType := mtError;
    3: DlgType := mtConfirmation;
  else {including 4:}
    DlgType := mtCustom;
  end;
  SoundMessageDlg (
    RadioGroup1.Items [RadioGroup1.ItemIndex],
    DlgType, [mbOK], 0);
end;
```

Láthatjuk tehát, hogy a SoundMessageDlg egy egyszerű, de igen hasznos függvény.

A csipogástól a szimfóniákig

Ha a MessageBeep függvényt használjuk, akkor csak az alapértelmezett Windows hangokat érjük el. Egy másik API eljárással, a PlaySound-dal nemcsak a beépített hangokat vehetjük igénybe, hanem tetszőleges WAV fájlt is lejátszhatunk. Ennek bemutatására is egy egyszerű programot készítettem. Ennek neve ExtBeep, futtatásának eredményét a 31.3. ábrán láthatjuk.



31.3. ábra *Az ExtBeep példaprogram formja*

A form listapaneljében néhány beépített hang és WAV fájl szerepel. Ezek ugyanabban a könyvtárban találhatóak, mint az ExtBeep.exe program. Amikor a felhasználó megnyomja a Play gombot, meghívjuk az MmSystem egységben definiált PlaySound eljárást:

```
procedure TForm1.PlayButtonClick(Sender: TObject);
begin
  PlaySound (PChar(Listbox1.Items[ListBox1.ItemIndex]),
    0, snd_Async);
end;
```

Az első paraméter a hang neve (ami egy Windows hang vagy pedig egy WAV fájl), vagy egy speciális hang erőforrás (ezt részletesebben a Win32 API súgójából ismerhetjük meg). A második paraméter megadja, hogy az előzőben megadott adat hol található meg. A harmadik paraméterrel pedig a lejátszás folyamatát szabályozhatjuk. Az előző példában azt állítjuk be, hogy a zene elindítása után azonnal visszakapjuk a vezérlést, és a hanganyag lejátszása a háttérben folyik. Ha a `snd_Sync` értéket adjuk meg, akkor a függvény csak akkor tér vissza, amikor a zene véget ért. Egy hosszú lejátszást megszakíthatunk a PlaySound függvény újbóli meghívásával, ekkor a `nil` értéket adjuk az első paraméternek:

```
procedure TForm1.StopButtonClick(Sender: TObject);
begin
  PlaySound (nil, 0, 0);
end;
```

Ez a pár sor akkor fut le az ExtBeep programban, amikor a felhasználó a Stop gombot nyomja meg. A Stoppal tudjuk leállítani a Loop gombbal elindított folyamatos lejátszást is. A Loop úgy hívja meg a PlaySound eljárást, hogy utolsó paraméterként a `(snd_Async or snd_Loop)` értéket kapja meg.

Alapértelmezett hangok használata

Egy Delphi alkalmazást futtatva észrevehetjük, hogy a fő form minimalizálásakor és visszaállításakor nem szólalnak meg az ezekhez az eseményekhez rendelt alapértelmezett Windows hangok. Ugyanakkor a teljes méretűre alakításkor illetve onnan visszaállításkor a hangok elhangzanak.

Megjegyzés

A Delphi alkalmazások a rejtett alkalmazásablak és a több form kezelésére használt belső, nem szabványos kód használata miatt nem adják ki az alapértelmezett hangokat.

Programjainkban a megfelelő hangok beállítása egyszerűen megoldható, és csak kevés változtatást kell elvégeznünk. Kezelnünk kell az Application objektum két ide tartozó eseményét, ezek az OnMinimize és az OnRestore. Itt megfelelő paraméterekkel meghívjuk a PlaySound API függvényt. Erre példát az AppSound alkalmazásban láthatunk. Ennek formja két új metódust tartalmaz:

```
type
  TForm1 = class (TForm)
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
  public
    procedure AppMini (Sender: TObject);
    procedure AppRestore (Sender: TObject);
  end;
```

Ezt a két metódust a FormCreate eljárásban hozzárendeljük a megfelelő eseményekhez:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMinimize := AppMini;
  Application.OnRestore := AppRestore;
end;

procedure TForm1.AppMini (Sender: TObject);
begin
  PlaySound ('Minimize', 0, snd_Async);
end;

procedure TForm1.AppRestore (Sender: TObject);
begin
  PlaySound ('RestoreUp', 0, snd_Async);
end;
```

Ezt a pár sort beillesztve programjainkba elérhetjük, hogy megszólaljanak a kívánt hangok. Ha végignézzük a Windows API súgófájlt, sok hasznos módszert találhatunk a multimédiás eszközök kezelésére.

A Médialejátszó komponens

Most térjünk vissza a Delphihez, és ismerkedjünk meg a Médialejátszó (Media Player) komponenssel. A `TMediaPlayer` osztály szinte mindent képes végrehajtani, amit a Windows médiakezelő felülete (Media Control Interface, MCI) lehetővé tesz. Ezekkel a módszerekkel a multimédiás eszközöket magas szinten programozhatjuk.

Talán a legfontosabb tulajdonsága ennek a komponensnek a DeviceType. Lehetséges értékei között szerepel a `dtAutoSelect` amely a FileName tulajdonságban megadott fájl kiterjesztésének megfelelő eszközt tételez fel, de

megadhatjuk pontosan is a hangkezelőt a megfelelő érték beállításával. A sok lehetőség közül talán legismertebbek a `dtAVIVideo`, a `dtCDAudio` és a `dtWaveAudio`.

Hogyha beállítottuk az eszköztípust és a fájlt is megadtuk, akkor megnyithatjuk az eszközt (vagy elegendő az AutoOpen tulajdonságot igazra állítanunk). Ezzel elérhetővé válnak a Médialejátszó nyomógombjai. A komponensen nem feltétlenül jelenik meg az összes gomb, amelyet az adott eszköz értelmezni tud. Három tulajdonság vonatkozik a gombokra. A VisibleButtons a megjelenő gombokat tartalmazza, az EnabledButtons az ezek közül engedélyezetteket, a ColoredButtons azokat, melyeken színes ábra jelenik meg. Az első kettővel véglegesen vagy ideiglenesen is szabályozhatjuk a gombok megjelenését, illetve elrejtését.

A komponens sok-sok eseményt kezel. Szokatlan az OnClick esemény, melynek két paramétere közül az első meghatározza, hogy melyik gombot nyomták le, a másodikkal pedig az alapértelmezett működést kikapcsolhatjuk. Az OnNotify esemény értesíti a komponenst arról, hogy sikeres volt-e a gomblenyomással elindított folyamat. Az OnPostClick esemény vagy a lejátszás kezdetekor vagy a végén következik be, a Wait tulajdonság beállításától függően. Ezzel a tulajdonsággal tudjuk szabályozni, hogy a lejátszás a háttérben (a program további működésével párhuzamosan) történik, vagy sem.

Hangfájlok lejátszása

Megint kezdjük egy nagyon egyszerű példával. Az MmSound program formján néhány címke-mezőben az aktuális állapotot figyelhetjük, egy nyomógombbal új fájlt adhatunk meg az OpenDialog komponens felhasználásával. A médialejátszó az alábbi beállításokkal szerepel:

```
object MediaPlayer1: TMediaPlayer
  VisibleButtons = [btPlay, btPause,
    btStop, btNext, btPrev]
  OnClick = MediaPlayer1Click
  OnNotify = MediaPlayer1Notify
end
```

Egy WAV vagy egy MIDI fájl megnyitása után a MediaPlayer gombjai elérhetővé válnak, és lejátszhatjuk a fájlt.

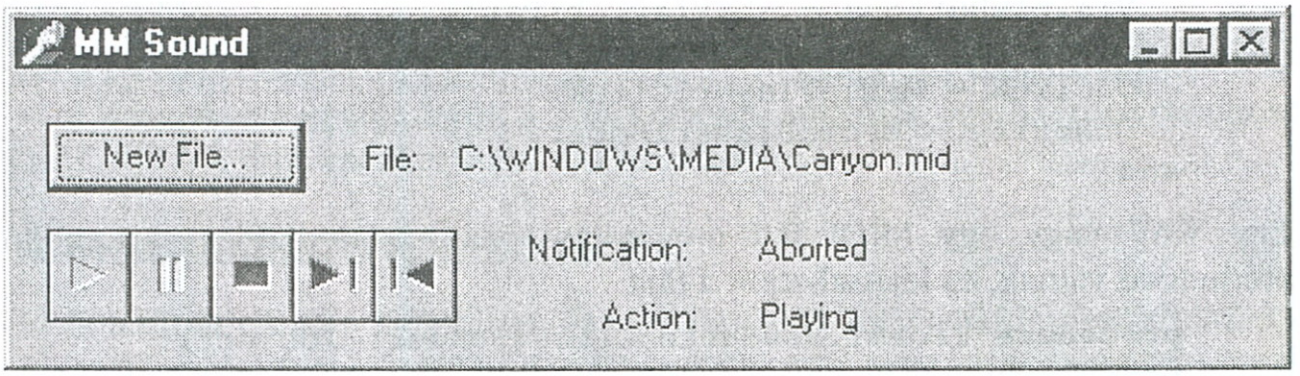
```
procedure TForm1.NewButtonClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    FileLabel.Caption := OpenDialog1.Filename;
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open; MediaPlayer1.Notify := True;
  end;
end;
```

A Notify tulajdonság igazra állításával érjük el, hogy a médialejátszó kiváltja a megfelelő eseményt, amely egy címke mezőben megjeleníti a kapott információt:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  case MediaPlayer1.NotifyValue of
    nvSuccessful : NotifLabel.Caption := 'Success';
    nvSuperseded : NotifLabel.Caption := 'Superseded';
    nvAborted    : NotifLabel.Caption := 'Aborted';
    nvFailure    : NotifLabel.Caption := 'Failure';
  end;
  MediaPlayer1.Notify := True;
end;
```

Ne felejtsük el a Notify tulajdonságot igazra állítani, ha az OnNotify eseménykezelőt meghívjuk, mert ha ezt nem tesszük, akkor a további üzeneteket nem kapjuk meg. Az egyik címke mezőben az utoljára kapott parancs olvasható:

```
procedure TForm1.MediaPlayer1Click(Sender: TObject;
  Button: TMPBtnType; var DoDefault: Boolean);
begin
  case Button of
    btPlay: ActionLabel.Caption := 'Playing';
    btPause: ActionLabel.Caption := 'Paused';
    btStop: ActionLabel.Caption := 'Stopped';
    btNext: ActionLabel.Caption := 'Next';
    btPrev: ActionLabel.Caption := 'Previous';
  end;
end;
```



31.4. ábra *Az MmSound program által megjelenített üzenet*

Videóklipek lejátszása

Eddig még csak hangokat használtunk. Itt az ideje, hogy a multimédia egy másik eszközét is megismerjük: a videóklipeket. Habár monitor minden gépen van, ahhoz, hogy videóklipeket (például AVI fájlokat) játszhassuk le, kifejezetten a Windows 95-höz beszerezhető meghajtóprogramra van szükség. Hogyha ez megvan, akkor innentől kezdve a videóklipeket használó program készítése ugyanolyan könnyű, mint hangok kezelése. Helyezzünk a formra egy médialejátszót, a FileName tulajdonsághoz válasszuk ki a megfelelő AVI fájlt, az AutoOpen tulajdonságot állítsuk igazra, és utána futtathatjuk programunkat. Amikor megnyomjuk a Play gombot, a Delphi egy külön ablakban lejátssza a videóklipet. Ez látható a 31.5. ábrán, ahol a Borland által terjesztett DELPHI.AVI látható.

31.5. ábra *Alapértelmezésben a videóklip egy külön ablakban fut*

Hogyha a klipet nem egy külön ablakban akarjuk megjeleníteni, akkor tegyünk a formra egy panelt (vagy bármilyen ablak jellegű komponenst), és ezt adjuk meg a médialejátszó Display tulajdonságában. A DisplayRect tulajdonsággal megadhatjuk, hogy az ablak mely részén jelenjen meg a klip. Az ezt bemutató MmVideo példát a 31.6. ábrán láthatjuk.

El lehet készíteni ezt a programot egyetlen sor kód begépelése nélkül is, de ehhez pontosan tudni kell, hogy hol és milyen AVI fájlok találhatók a számítógépen. Ezért inkább megírtam azt a rövid kis rutint, amellyel megnyithatunk fájlokat, és azokat azonnal le is játszhatjuk őket. Ehhez csak kattintanunk kell a panelen, ahogy a feliratból ki is derül.

```
procedure TForm1.Panel1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    MediaPlayer1.FileName := OpenDialog1.Filename;
    MediaPlayer1.Open;
    MediaPlayer1.Perform(wm_LButtonDown, 0, $00090009);
    MediaPlayer1.Perform(wm_LButtonUp, 0, $00090009);
  end;
end;
```

A médialejátszó megnyitása után azonnal meghívhatnánk a Play metódust, de ekkor a gombok engedélyezése és letiltása nem megfelelően lenne beállítva. Ezért inkább egy kattintást szimulálunk a médialejátszó ablakának (9,9) koordinátájú pontjában (a 32-bites értéket egy függvényhívással is elő lehetett volna állítani, de használhatjuk a fenti példában látható hexadecimális megadási módot is). A hibák elkerülése érdekében az összes gombot letiltottam a szimulált gomblenyomásig. Ugyancsak automatikusan leállítjuk a lejátszást, amikor a program véget ér (az OnClose esemény-kezelővel).

31.6. ábra *Videóklip egy panelben. Hasonlítsuk össze ezt az ábrát az előzővel, ahol a klip az alapértelmezett ablakban jelenik meg!*

Videóklip a formon

A médialejátszó komponens csak bizonyos követelményeknek eleget tevő ablakokban képes megjelenni. Sok komponens használata engedélyezett, de nem mindegyiké. Furcsa ötlet magát a Médialejátszó komponenst használni a videóklip kimeneti ablakának. Bár ez elvileg lehetséges, két problémába is ütközünk. Az első az, hogy nem helyezhető el akárhol, és nem is méretezhetjük szabadon ezt a komponenst. Ha nagy gombokat próbálunk használni, ezek

mérete automatikusan lecsökken futtatáskor. A másik probléma, hogy ha a Pause gombra kattintunk, akkor ez a videóklip előtt látszik, míg a többi gomb el lesz fedve (azért tanulságos kipróbálni ezt a lehetőséget is).

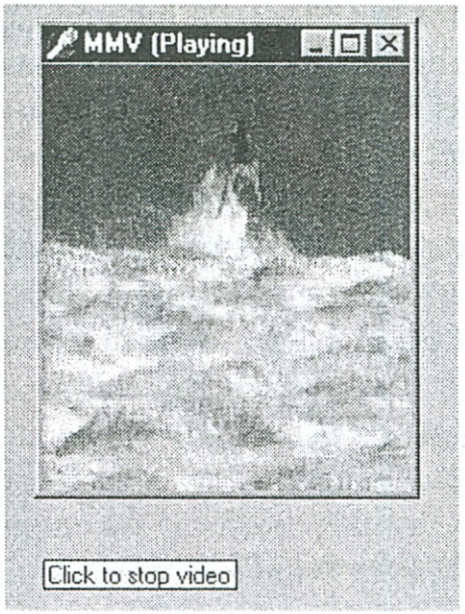
Elég nehézkes a videóklipet egy formon megjeleníteni. Tervezéskor a médialejátszó Display tulajdonságában nem tudunk formot beállítani, de futtatáskor már lehetséges ez az érték. Ennek kipróbálásához helyezzünk egy rejtett (hamisra állított Visible tulajdonságú) médialejátszót és egy OpenDialog komponenst a formra. Állítsuk be megfelelően a form címét és egy tippet, majd engedélyezzük a ShowHints tulajdonságot. Ezek után a következő kód használatával elérhetjük, hogy a formra való kattintás a klip betöltését és lejátszását eredményezze:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  if MediaPlayer1.FileName = '' then
    if OpenDialog1.Execute then
    begin
      MediaPlayer1.FileName := OpenDialog1.FileName;
      MediaPlayer1.Open;
      Playing := False;
    end
    else
      exit; // stop if no file is selected
  if Playing then
  begin
    MediaPlayer1.Stop;
    Playing := False;
    Caption := 'MM Video (Stopped)';
    Hint := 'Click to play video';
  end
  else
  begin
    MediaPlayer1.Display := self;
    MediaPlayer1.DisplayRect := ClientRect;
    MediaPlayer1.Play; Playing := True;
    Caption := 'MMV (Playing)';
    Hint := 'Click to stop video';
  end;
end;
```

Ebben a rutinban a Playing a form egy privát logikai adata. Vegyük észre, hogy az egész formot felhasználjuk a klip megjelenítésére. Ha a felhasználó átméretezi a formot, akkor az alábbi sorokkal igazítjuk a képet az ablakhoz:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  MediaPlayer1.DisplayRect := ClientRect;
end;
```

Bár a videóklip megjelenítése valószínűleg eredeti méretében a legjobb, ezzel a programmal tetszőlegesen átméretezhetjük, ahogy ez a 31.7. ábrán is látható. A túlzott méretezés természetesen rontja a kép minőségét.



31.7. ábra *Egy átméretezett videóklip (nagyon rossz felbontásban) és a gomb tippje*

A médialejátszó akkor is megáll, ha a fájl véget ér, vagy valamilyen hiba keletkezik. Mindkét esetben egy Notify esemény keletkezik:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  Playing := False;
  Caption := 'MMV (Stopped)';
  Hint := 'Click to play video';
end;
```

A CD meghajtó használata

Az audio és video fájlok kezelésén kívül az MCI kezelőfelület más külső eszközök kezelését is támogatja. A sok-sok elem közül valószínűleg a leggyakrabban használt a CD-ROM. A legtöbb CD audio lemezeket is tud olvasni, és a zenét egyenesen a hangkártyának küldik. Az MCI interfésszel és a médialejátszóval ilyen eszközöket használó programot is írhatunk. Mindenekelőtt `dtCDAudio` értékre kell állítanunk a DeviceType tulajdonságot, ellenőriznünk kell, hogy a FileName tulajdonságban ne legyen file kijelölve, és a CD-lejátszónak is készen kell állnia.

Ha elhelyezünk egy médialejátszót egy formon, elvégezzük a fenti beállításokat, lefordítjuk és futtatjuk programunkat, azzal egy teljes zenei CD lejátszót készítettünk. Amikor elkezdjük szépítgetni a lejátszónkat, tapasztalhatjuk, hogy mégsem olyan egyszerű az élet, mint amilyennek első látásra tűnik. Most egy olyan példát mutatunk, mely néhány extra lehetőséget is biztosít, kihasználva a Windows CD-kezelését. A program formja tartalmaz néhány nyomógombot, néhány címkét az aktuális állapot visszajelzésére és egy SpinEdit komponenst, ahol kiválaszthatjuk a CD-ről a kívánt számot.

A címkéket arra használjuk, hogy tájékoztassuk a felhasználót a CD-n található dalok számáról, az éppen lejátszás alatt lévő zene hosszáról és az aktuális pozícióról az időzítő felhasználásával.

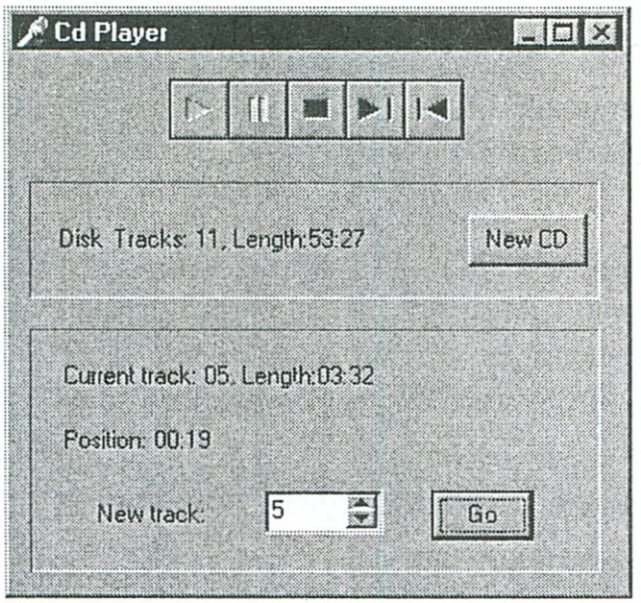
Általában, ha lehetőség van rá, használjuk a tfTMSF (Track, Minute, Second, Frame) értéket a médialejátszó TimeFormat tulajdonságában, hogyha pozíciót használó tulajdonságokat (ilyenek a Position és a Length) akarunk kezelni. Ezek elérése viszonylag egyszerű, ha az MmSystem egység megfelelő függvényeit használjuk. Például:

```
CurrentTrack := Mci_TMSF_Track (MediaPlayer1.Position);
```

Az alábbi két eljárás segítségével számíthatjuk ki a teljes CD és a lejátszás alatt álló szám hosszát:

```
procedure TForm1.CheckDisk;
var
  NTracks, NLen: Integer;
begin
  NTracks := MediaPlayer1.Tracks;
  NLen := MediaPlayer1.Length;
  DiskLabel.Caption := Format (
    'Tracks: %.2d, Length:%.2d:%.2d', [NTracks,
    Mci_TMSF_Minute (NLen), Mci_TMSF_Second (NLen)]);
  SpinEdit1.MaxValue := NTracks;
end;
procedure TForm1.CheckPosition;
var
  CurrentTrack, CurrentPos, TrackLen: Integer;
begin
  CurrentPos := MediaPlayer1.Position;
  CurPosLabel.Caption := Format ('Position: %.2d:%.2d',
    [Mci_TMSF_Minute (CurrentPos),
    Mci_TMSF_Second (CurrentPos)]);
  CurrentTrack := Mci_TMSF_Track (CurrentPos);
  TrackLen := MediaPlayer1.TrackLength [CurrentTrack];
  TrackNumberLabel.Caption := Format (
  'Current track: %.2d, Length:%.2d:%.2d',[CurrentTrack,
  Mci_MSF_Minute(TrackLen), Mci_MSF_Second(TrackLen)]);
end;
```

A kód csak a sok típuskonverzió miatt tűnik bonyolultnak. Megjegyezzük, hogy az aktuális szám hosszát (amit a TrackLength tulajdonságban találhatunk) nem az alapértelmezett formában jelenítjük meg, mint ahogy azt a Súgó javasolja, hanem az MSF (Minute Second Frame) formátummal. Ennek eredménye a 31.8. ábrán látható.



31.8. ábra *A CDPlayer példa, futás közben*

A teljes CD-re vonatkozó adatokat csak a New CD gomb lenyomásakor számoljuk ki:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MediaPlayer1.TimeFormat := tfTMSF;
  MediaPlayer1.Open;
  CheckDisk;
  CheckPosition;
end;

procedure TForm1.NewButtonClick(Sender: TObject);
begin
  CheckDisk;
  CheckPosition;
end;
```

Az aktuális számra és a pozícióra vonatkozó értékeket az időzítő minden ütemében frissítjük a CheckPosition metódus segítségével. Ez messze nem tökéletes megoldás, hiszen ha más programokat is használni akarunk zenei CD-nk lejátszása közben, a médialejátszó állandó lekérdezése túlságosan lelassíthatja

a rendszert. Ez természetesen lényegesen függ a hardverkörnyezettől. Azon kívül, hogy tájékoztatjuk a felhasználót arról, hogy mi történik, a médialejátszó gombjaival elindíthatja, leállíthatja a lejátszást, más számot választhat ki, stb. Ezek a műveletek megállítják, majd újraindítják az időzítőt:

```
procedure TForm1.MediaPlayer1PostClick(
  Sender: TObject; Button: TMPBtnType);
begin
  if MediaPlayer1.Mode = mpPlaying then
    Timer1.Enabled := True
  else
    Timer1.Enabled := False;
  CheckPosition;
end;
```

A SpinEdit komponensben kiválasztott zenére a Go nyomógombbal is ugorhatunk. A SpinEdit MaxValue tulajdonságát a CheckDisk metódus állítja be.

```
procedure TForm1.GoButtonClick(Sender: TObject);
var
  Playing: Boolean;
begin
  Playing := (MediaPlayer1.Mode = mpPlaying);
  if Playing then
    MediaPlayer1.Stop;
  MediaPlayer1.Position :=
    MediaPlayer1.TrackPosition[SpinEdit1.Value];
  CheckPosition;
  if Playing then
    MediaPlayer1.Play;
end;
```

Programunkat továbbfejleszthetjük egy CD adatbázissal, amely az összes CD lemezünk címét és a rajtuk található összes zeneszám előadóját és címét tartalmazza – ezt már én is régen megírtam volna, ha lenne elég időm CD-im adatainak begépelésére... Hasonló program egyébként létezik már Windows 95 alatt.

Utószó

Ebben a fejezetben megvizsgáltuk, hogyan lehet zenei és videó eszközöket használó programokat készíteni. Láttuk, hogyan rendelhetünk hozzá hangokat felhasználói eseményekhez. Néztünk példát hangfájlok, videófájlok és a CD használatára a médialejátszó segítségével. Ahogy a számítógépek és a CD-ROM-ok évről-évre gyorsabbá válnak, egyre több programunkban kihasználhatjuk a multimédia által nyújtott lehetőségek széles tárházát. Ne becsüljük alá a könnyű programozhatóságot ezen a területen sem, még akkor sem, hogyha már komoly üzleti programokat írunk.

A Delphi egy csodálatos programozási környezet. Most, hogy a könyv végére értünk, bátran próbálgassuk képességeinket és a Delphi lehetőségeit. Ahogy már a bevezetésben említettem az én (Angol nyelvű) és a Kiadó (Magyar nyelvű) honlapján találhatunk javításokat, módosításokat a könyvvel kapcsolatban, az alábbi címeken:

```
http://ourworld.compuserve.com/homepages/marcocantu
http://www.kiskapu.hu/konyvek/delphi
```

Az én honlapomról további Delphi anyagokat is elérhetünk: teljes dokumentációt és újabb komponenseket. További hivatkozásokat találhatunk a CD mellékleten.

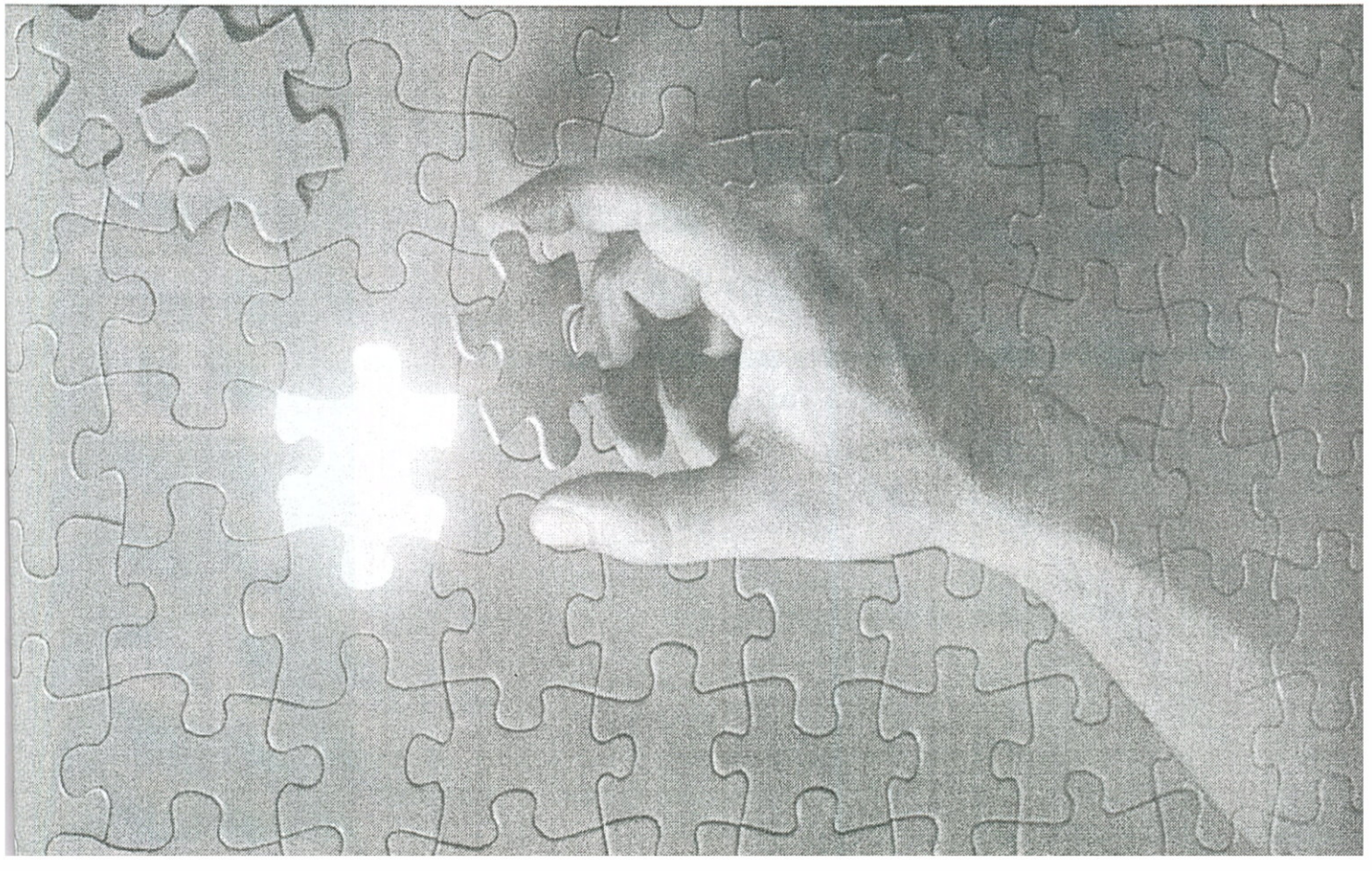
Remélem, könyvem elolvasása legalább annyi örömöt jelentett, mint amennyit az elkészítése. Bármilyen angol nyelvű kérdést, kiegészítést szívesen fogadok e-mail címemen:

```
marcocantu@compuserve.com
```

A Kiskapu Kiadó örömmel nyújt felvilágosítást magyar nyelven:

```
kiskapu@kiskapu.hu
```

Függelékek



A. FÜGGELÉK

Az OOP szemlélet rövid áttekintése

E fejezet tartalma:

- Absztrakció a programozási nyelvekben
- Osztályok
- Öröklődés
- Polimorfizmus
- Az OOP egy definíciója
- Az OOP nyelvek

Az OOP szemlélet rövid áttekintése

A Delphi nyelve, az Object Pascal egy objektum-orientált programozási nyelv. Az 5. fejezetben megismerhettük a nyelv objektum-orientált lehetőségeit, de nem sokat szóltunk még az OOP szemléletről általánosságban. Ez a függelék egy rövid áttekintést kíván nyújtani az OOP legfontosabb elemeiről.

Absztrakció a programozási nyelvekben

Az OOP alapja az absztrakció. A programozási nyelveket azért készítik és azért használják, hogy az eljárásokat a processzor gépi kódjánál magasabb szinten írhassuk le. Általában a programozási nyelvek által nyújtott absztrakciós lehetőségeket két típusra oszthatjuk aszerint, hogy az adatstruktúrák vagy a vezérlési szerkezetek megvalósításakor lépnek fel.

Az első lépés az adatreprezentációs absztrakció felé maga az adattípus fogalma. Az első generációs programozási nyelvek bevezették a típus fogalmát. A második generációs nyelvek (köztük a Pascal) lehetőséget adtak a felhasználónak olyan saját típusok definiálására, mint például a rekord vagy a tömb. Később az adattípus továbbfejlődésével kialakult az absztrakt adattípus fogalma, amely az ábrázolt adatot összekapcsolta az ezt felhasználó, módosító műveletekkel. Egy absztrakt adattípusban általában az adatok fizikai ábrázolása a felhasználó elől rejtve van. Ezekben a nyelvekben az új adattípusokat a nyelv által nyújtott típusok kiterjesztéseként foghatjuk fel. Az OOP nyelvek osztályait az absztrakt adattípusok továbbfejlesztésének tekinthetjük.

A vezérlési szerkezetek szempontjából az első programozási nyelvek ismerték az ugró utasításokat, az elágazásokat (mint például a `case` vagy az `if-then-else` szerkezet), valamint a ciklusokat (mint a `for` és a `while`). Később már szubrutinokat is használhattunk, eljárások és függvények formájában. A szubrutinok megjelenésével egyszerre sokféle programozási modellt kifejlesztettek, mint például felülről lefelé történő probléma dekompozíciót (functional top-down decomposition). A szubrutinok absztrakciója abban nyilvánul meg, hogy bárhol meghívhatjuk őket, tudjuk, hogy mit csinálnak, de nem kell tudnunk, hogyan végzik el feladatukat. Akárcsak az absztrakt adattípusokat, az új szubrutinokat is a nyelv kiterjesztéseinek tekinthetjük.

Osztályok

Az első és legfontosabb fogalom, amivel meg kell barátkoznunk az OOP megismerésekor az osztály. Az osztály objektumok egy csoportjának absztrakt leírása. Ezen objektumok mindegyikének saját állapota van, de ugyanazokat a műveleteket végezhetjük velük.

Például egy palack egy objektum. Vannak állapotai: lehet üres vagy teli, és különböző dolgokat tartalmazhat, amit általában egy feliratról tudhatunk meg; és

bizonyos műveleteket végezhetünk vele: feltölthetjük-kiüríthetjük, megváltoztathatjuk a tartalmát, kinyithatjuk-bezárhatjuk, stb. Természetesen néhány művelet függ az objektum aktuális állapotától, például egy üres üveg kiürítésének nem sok értelme van. Más szóval, az osztály egy típus, amelynek van néhány adattagja (ezek írják le az osztály egy objektumának állapotát), és néhány metódusa (a vele végezhető műveletek). A műveletek viselkedése függhet az objektum aktuális állapotától.

Megjegyzés

Jegyezzük meg, hogy az OOP-ben az osztály egy adattípust jelent, az objektum az adattípus egy példányát, egy változót. Sajnos, néhány OOP nyelv (köztük az Object Pascal korábbi verziói is) más terminológiát használ.

Egy programban az osztályoknak két fő célja van:

- Az osztály definiálja az absztrakt adattípust (adattagok és műveletek). Az osztályokat használhatjuk a valós világ bármilyen bonyolultságú egyedeinek leírására. Ha egy nagyon összetett objektumot akarunk leírni, akkor használhatunk osztályokat az objektum alobjektumainak, alrendszereinek leírására is. Ahogy már említettem, az osztályokat a program adattípusainak kiterjesztéseként foghatjuk fel. Ennek megfelelően, a saját magunk által készített új Delphi komponenseket a Delphivel együtt terjesztett VCL kiterjesztésének tekinthetjük.
- Az osztályok a program alapegységei. Az Object Pascalban minden új osztályt (egy form, vagy más elemek definiálásakor) a saját egységében helyezhetünk el. Ezzel még a nagy, összetett rendszereket is kisebb, jól kezelhető darabokra oszthatjuk. Ehhez a megközelítéshez igazodik a formok forráskódjának kezelése a Delphiben.

Ahhoz, hogy egy nyelvet objektum-orientáltnak nevezhessünk nem elég, hogy az osztályok jelentsék alapegységeit. Két további fontos jellemző az öröklődés és a polimorfizmus.

Öröklődés

Az öröklődéssel új osztályokat tudunk készíteni már definiált osztályok felhasználásával. Tehát nem kell mindig teljesen új osztályokat építenünk, hanem örököltethetünk adatmezőket és műveleteket a szülő osztálytól.

Hogy példánkat folytassuk, az általános palack osztályból származtathatunk speciálisakat, például boros- vagy egy sörösüveget. Minden egyes palacknak vannak rá jellemző tulajdonságai, különböző módon kell kezelnünk őket. Azonban vannak olyan tulajdonságaik, műveleteik, melyek minden palackra jellemzőek.

Az öröklődés erőteljes programozási eszköz, de gyakran nem könnyű megérteni, és hatásosan használni. Néhány általánosan felhasználható jó tanács:

- Az öröklődés gyakorlatilag specializációt jelent. A gyermek-osztályt tekinthetjük a szülő-osztály egy speciális esetének. Ezt a viszonyt gyakran fejezzük ki „az egy" szókapcsolattal, ahogy például a borospalack *az egy* palack (angol nyelvű szakkönyvek erre a viszonyra az „is a" szópárt használják).
- Bizonyos esetekben az öröklődést használhatjuk általánosításra is. Tegyük fel, hogy van egy menedzser, egy titkár és egy könyvelő osztályunk. Ekkor létrehozhatunk egy általános alkalmazott osztályt, amit az előző három közös szülőjének tekinthetünk. Ezt az osztályt használhatjuk a fentiek közös tulajdonságainak és műveleteinek megfogalmazására, ezzel programunkat strukturáltabbá, áttekinthetőbbé tehetjük.
- Gyakorlatban az öröklődés lehetőséget ad a programjainkban lévő felesleges ismétlődések elkerülésére. Hasonló osztályok készítésekor programsorok másolása helyett sokkal egyszerűbb az egyiket a többi szülőjeként kezelni. Ezzel a technikával nemcsak hogy rövidebbé tehetjük programunkat, de a tesztelési időt is lerövidítjük, és a későbbi módosításokat is egyszerűbbé tesszük, azáltal, hogy a kódrészlet csak egyszer szerepel a programban.
- Még egy érv az öröklődés használata mellett, hogy a fordító felismeri és támogatja a szülő-gyermek viszonyt, és kihasználhatjuk az ezzel járó előnyöket. Ezeket nézzük meg a következő részben

Polimorfizmus

A harmadik kulcsfontosságú elem az OOP nyelvekben a polimorfizmus (többalakúság). Ez szó szerint az objektumoknak azt a képességét jelenti, hogy sokféle alakban jelenhetnek meg. Lehetővé teszi, hogy különböző osztályok objektumaira ugyanazon a változón keresztül hivatkozzunk. Attól függően, hogy a változó éppen melyik osztályba tartozó objektumot tartalmaz, ugyanazt a műveletet más-más módon hajtjuk végre. Más szóval, ha végrehajtunk egy műveletet egy objektumon, akkor az a típusának megfelelően válaszolhat.

Például, deklaráljunk egy általános palack objektumot tartalmazó változót (nevezzük ezt MyBottle-nek). Töltsük fel egy `TWineBottle` vagy egy `TBeerBottle` osztályba tartozó objektummal (a `T` kezdőbetű egy általános

Borland programozási konvenció a `Type` szó rövidítéseként). Legyen mindegyik osztálynak egy Open metódusa, mindegyikben más-más megvalósítással. Tételezzük fel, hogy ez a metódus dinamikusan kötött, ami azt jelenti, hogy a fordító nem határozza meg, hogy melyik eljárást hívja meg, erről döntés csak futási időben történik. Az Object Pascalban ezt úgy érhetjük el, hogy a metódust a szülő osztályban virtuálisnak vagy dinamikusnak deklaráljuk (5.fejezet), és a származtatott osztályokban az `override` kulcsszóval definiáljuk át.

Ezek után mi történik, ha a MyBottle Open metódusát hívjuk meg? Természetesen az objektum aktuális típusának megfelelő eljárás fut le. Hogyha a MyBottle jelenleg `TBeerBottle` típusú, akkor kinyitásához sörnyitót kell használnunk. A borosüveg kinyitásához merőben más módszerekre (esetünkben dugóhúzóra) van szükség.

Az öröklődés és a polimorfizmus együttes használatával programunkat osztályhierarchiára alapozhatjuk. Az OOP szemlélete az osztályhierarchia készítésén és bővítésén alapul. Ezt tükrözik a Delphi komponensek is.

Az OOP egy definíciója

Most, hogy megismertük az osztályokat, az öröklődést és a polimorfizmust definiálhatjuk az OOP nyelveket. Egy nyelv objektum orientált, ha legalább a fenti három eszköz használatára lehetőséget ad. Precízebben, nem nevezhetünk egy nyelvet objektum orientáltnak, ha az alábbi három kritériumot nem teljesíti:

- Objektumokat hozhatunk létre, melyeknek állapotaik és műveleteik vannak.
- Az objektumok osztályok példányai. Az osztályok definiálják a műveleteket és az adatreprezentációt.
- Támogatják az öröklődést és a polimorfizmust, melyek szorosan összefüggenek.

Vannak ennél összetettebb és teljesebb OOP definíciók is, de ezek mindig felhasználják egy-egy nyelv speciális lehetőségeit.

Az OOP nyelvek

A Simula, az első nyelv ami ismerte az osztályok fogalmát, már 1967-ben megjelent. A Smaltalk 1980 körül már megvalósította az OOP fő eszközeit. Ezután hibrid nyelvek következtek (melyek létező nyelvekben valósították meg az objektumokat), ilyenek a C++, az Objective-C, CLOS és az Object Pascal. Megemlíthetjük az Eiffelt és a Sathert is, de folyamatosan jelennek meg újabb és újabb OOP nyelvek, mint például a Java.

Az OOP nyelvek három szempont szerint csoportosíthatjuk. Tiszta vagy hibrid, dinamikus vagy statikus típusellenőrzéses, és hagyományos vagy referenciákat használó nyelvek.

Tiszta és hibrid OOP nyelvek

- A tiszta OOP nyelvek nem teszik lehetővé más programozási modell használatát. Ezekben a nyelvekben nem írhatunk különálló eljárásokat, csak osztályon belüli metódusok fordulhatnak elő. Ugyanígy nem lehetnek globális változóink sem, hanem osztály-adattagokat hozunk létre. Erre példa a Smalltalk vagy a Java.
- Hibrid nyelvekben ezzel szemben bármit megtehetünk, akár teljesen el is felejthetjük az objektumokat. Ezek egy már létező nyelv továbbfejlesztéseként jöttek létre. Ilyen a C++ vagy az Object Pascal.

Statikus és dinamikus típusellenőrzés

- A statikus típusellenőrzést használó nyelvek az adattípus fogalmán alapulnak, és a lehető legtöbb típusellenőrzést végzik el már fordítási időben. Az Object Pascal, a C++ és a Java is szigorúan típusos OOP nyelv. Ezekben a nyelvekben egy fordítóprogram készíti el a futtatható kódot.
- A dinamikus típusellenőrzést használó nyelvek a típusnak egy gyengébb definícióját követelik meg. A típusellenőrzések többségét futásidőben végzik. Ezek a nyelvek általában interpretert hazsnálnak, mint például a Smalltalk.

Tradicionális és referenciaelvű objektummodell

- Sok tradicionális OOP nyelv, különösképpen a C++, lehetővé teszi, hogy objektumokat hozzunk létre a globális memóriaterületen, a veremben vagy a heapen. Az objektumok veremben történő létrehozása azt jelenti, hogy a rendszer kicsit egyszerűbben kezeli az objektumok helyfoglalását és felszabadítását, de ekkor a polimorfizmus megvalósítása nehézkesebb.
- Az OOP nyelvek nagyobbik csoportja, például az Object Pascal vagy a Java, egy objektum-referencia modellt valósítanak meg. Minden objektum helyfoglalása dinamikusan történik (ez általában egy – esetleg explicit – konstruktorhívással valósul meg). Java-ban nincs szükség az objektumok megszüntetésére, mert ezt elvégzi helyettünk egy „szemétgyűjtő" eljárás (garbage collector). Delphiben a legelegánsabb megoldás a feleslegessé vált objektumok helyének automatikus felszabadítására a komponensek tulajdonviszonyait használja fel.

Objektum-orientált elemzés és tervezés

Az OOP nyelveket sokféleképpen használhatjuk, főképp, ha egy létező nyelv kiterjesztései. Azonban ahhoz, hogy teljes mértékben kiaknázhassuk az OOP lehetőségeit, az objektum-orientált elemzés és tervezés módszereivel kell megközelítenünk a megoldandó problémát. Számos módszertanát kidolgozták az objektum-orientált analízisnek (Object-Oriented Analysis, OOA) és az objektum-orientált tervezésnek (Object-Oriented Design, OOD). Ezek részletes tárgyalása külön kötetet igényelne, ezért itt csupán azt szeretnénk kiemelni, hogy az objektum-orientált programozás nemcsak egy megfelelő nyelvet és programozási környezetet igényel, hanem egy helyes szemléletet is. A téma iránt érdeklődők bőséges irodalomból válogathatnak.

Ne becsüljük alá az OOP szerepét a Delphiben. Szánjunk egy kis időt legalább az OOP alapjainak megismerésére. Ha már van egy kis rálátásunk az OOP-re, akkor jobban megértjük a Delphi belső felépítését, és jobb programokat írunk.

Az osztályok és a komponensek nagyszerű eszközt nyújtanak az újrafelhasználhatósághoz, de a megfelelő tervezés nélkül bármely összetettebb rendszer teljesen áttekinthetetlenné válhat. Ahhoz, hogy kihasználhassuk az OOP nyújtotta előnyöket (mint pl. az újrafelhasználhatóság és a könnyű karbantarthatóság), tisztában kell lennünk az OOP technikákkal, ezt próbáltuk ebben a könyvben bemutatni. Természetesen az első lépés a nyelv megismerése, ehhez nyújt segítséget az 5. fejezet, ami részletesen tárgyalja az Object Pascalt.

B. FÜGGELÉK

Bevezetés az SQL-be

E fejezet tartalma:

- Mi az SQL?
- A Select utasítás
- Ismétlődő rekordok kiküszöbölése
- Kapcsolatok létrehozása
- A listázási sorrend megváltoztatása
- Számított értékek
- Csoportok definiálása
- További SQL utasítások

Bevezetés az SQL-be

Amikor a Query komponenst használjuk a Delphiben adatbázis alkalmazásokhoz, a lekérdezés definiálásához egy SQL utasítást kell írnunk a komponens SQL tulajdonságához, mint ahogy a 16. fejezetben láttuk. Ha a Delphi Client/Server verzióját használjuk, akkor az SQL utasítások közvetlen beírása helyett igénybe vehetjük a Visual Query Builder eszközt. Mindkét esetben ismernünk kell az SQL nyelvet. A 16. és 17. fejezetben már láttunk néhány egyszerű SQL utasítást, de ott nem néztük meg részletesen ezek felépítését.

Ez a függelék azt a célt szolgálja, hogy megismertessük az SQL használat alapjait azokkal, akik még soha nem használták. A Database Desktoppal kipróbálhatjuk az itt szereplő parancsokat, ha beállítjuk a megfelelő adatbázis nevet (DBDEMO) az **SQL** menü **Alias** pontjával.

Mi az SQL?

Mielőtt végignéznénk az utasításokat, és megvizsgálnánk, hogyan működik az SQL, szükségünk van az SQL definíciójára. Az SQL mozaikszó, a Structured Query Language (Strukturált Lekérdező Nyelv) rövidítése. Az SQL a relációs adatbázis-kezelő rendszerek (Relational Database Management System, RDBMS) készítéséhez és kezeléséhez leggyakrabban használt nyelv. Nagy előnye, hogy sokféle hardver-környezetben és platformon használható és szabványnak tekinthető, bár a különböző megvalósítások között jelentős eltérések lehetnek.

Itt a nyelvnek csupán néhány elemét tekintjük át, melyek mindegyik SQL nyelvjárásban megtalálhatók. Megjegyzendő, hogy bár az SQL név lekérdezésekre utal, a nyelv lehetőséget ad az adatbázis frissítésére, rekordok törlésére, hozzáadására is.

A Select utasítás

A legfontosabb parancs az SQL-ben minden valószínűség szerint a `select`. Három fő részre bontható:

- A `select` szó után megadjuk a lekérdezendő mezőket. Ha a tábla összes mezőjét szeretnénk felhasználni, akkor a mezőlista helyett elegendő egy csillagot (*) írnunk.
- A `from` részben azokat a táblákat kell felsorolnunk, amelyek a lekérdezésben részt vesznek.
- A `where` részben az eredménytáblába bekerülő rekordokat választhatjuk ki, szűrőfeltételek megadásával. Ez a rész el is hagyható, ekkor az összes rekord szerepel az eredményben.

A következő utasítás például a COUNTRY.DB tábla minden sorát (rekordját) kiválasztja:

```
select * from Country
```

Az alábbi paranccsal csak a megadott oszlopok (mezők) jelennek meg a lekérdezésben:

```
select Name, Capital from Country
```

Ez a sor pedig csak azokat a rekordokat listázza, melyek teljesítik a megadott feltételt:

```
select Name, Capital
from Country
where Population > 20000000
```

Az eredménytáblában csak azoknak az országoknak a neve és fővárosa szerepel, melyek népessége 20 milliónál nagyobb. A `where` részben sokféle egyszerű feltételt használhatunk, de lehetőségünk van ezek összefűzésére is. Ha az `and` szóval kapcsoljuk össze a feltételeket, akkor mindegyiknek teljesülnie kell az aktuális rekordra, ahhoz, hogy az megjelenjen az eredményben. Az `or` használatával már az egyik feltétel teljesülése is elegendő ehhez:

```
select *
from Country
where Continent = "South America"
  and Population < 10000000

select Name, Capital
from Country
where Name <= "Brazil"
  or Capital >= "Ottawa"
```

Az első parancs a 10 milliónál kisebb lakosságú dél-amerikai országokat sorolja fel, míg a második azokat, melyeknek neve abc szerint a „Brazil" előtt van és azokat melyek fővárosa „Ottawa" után szerepel az ábécérendben.

Ismétlődő rekordok kiküszöbölése

Próbálkozzunk a kontinensek kiíratásával a Country tábla segítségével:

```
select Continent from Country
```

Ez fel is sorolja az összes földrészt, sőt mindegyiket annyiszor, ahány ország van rajta. Az ismétlődések (amikor a megadott mezők több rekordban azonos értéket tartalmaznak) elkerüléséhez egy új SQL kulcsszót használunk, ez a `distinct`:

```
select distinct Continent from Country
```

Kapcsolatok létrehozása

Sok esetben az utasítások több táblára hivatkoznak. Ha két táblát szeretnénk használni, akkor nem feltétlenül kell összekapcsolnunk őket, hanem két `select`-tel megvalósítva a lekérdezést az első eredménytábláját a második feltételében használhatjuk fel. Ha viszont több táblával dolgozunk, mindenképpen érdemes összekapcsolnunk őket. Az SQL-ben általában nincs külön parancs ennek megvalósítására, bár egyes megvalósítások, mint például az InterBase vagy az Access, bevezetnek külön kulcsszót e célból.

Két táblázatot egyszerűen összekapcsolhatunk úgy, hogy a `where` részben két tábla egy-egy mezőjének egyenlőségét követeljük meg. Az Orders (rendelések) és a Customer (vevő) táblák összekapcsolásával kiírathatjuk egy cég összes megrendelésének dátumát. Ehhez írjuk be a következő utasítást:

```
select Orders.OrderNo, Orders.SaleDate,
  Customer.Company, Customer.Contact
from Orders, Customer
where Orders.CustNo = Customer.CustNo
```

Ehhez hasonlóan, két feltétellel, három tábla összekapcsolására is lehetőségünk van. Ha az előbbi példában arra is kíváncsiak vagyunk, hogy melyik alkalmazott vette fel a megrendelést:

```
select Orders.OrderNo, Orders.SaleDate,
  Customer.Company, Customer.Contact, Employee.LastName
from Orders, Customer, Employee
where Orders.CustNo = Customer.CustNo
  and Orders.EmpNo = Employee.EmpNo
```

A következő SQL parancs segítségével a vevők azon megrendeléseinek értékét kapjuk meg, melyeket Johnson bonyolított le:

```
select Orders.AmountPaid, Customer.Company
from Orders, Customer, Employee
where Orders.CustNo = Customer.CustNo
  and Orders.EmpNo = Employee.EmpNo
  and Employee.LastName = "Johnson"
```

A listázási sorrend megváltoztatása

A `select` utasítást kiterjesztve az `order by` résszel, lehetőségünk van az eredménytábla rekordjainak sorrendjét megváltoztatni. Az alábbi parancs eredményében a rekordok elsődlegesen a földrész, ezen belül az ország neve szerint rendezetten jelennek meg.

```
select *
from Country
where Population > 10000000
order by Continent, Name
```

Számított értékek

A `select` lehetőséget ad arra is, hogy több rekord felhasználásával számított értéket írassunk ki. Például az alkalmazottak számát így kaphatjuk meg:

```
select count (*) from Employee
```

Hogy egy bonyolultabb példát is mutassunk, számítsuk ki egy adott alkalmazott összes megrendeléseinek számát, összértékét és átlagát:

```
select  Sum (Orders.AmountPaid),
        Count (Orders.AmountPaid),
        Avg (Orders.AmountPaid)
from Orders, Employee
where Orders.EmpNo = Employee.EmpNo
  and Employee.LastName = "Johnson"
```

Csoportok definiálása

Számított mezők használatakor a rekordok csoportokba rendezésével az egyes értékeket csoportonként számolhatjuk. A csoportosítás szempontját a `group by` kulcsszó után adhatjuk meg. A következő lekérdezés felsorolja az alkalmazottakat és az általuk lebonyolított üzletek összértékét.

```
select Employee.LastName, Sum(Orders.AmountPaid)
from Orders, Employee
where Orders.EmpNo = Employee.EmpNo
group by Employee.LastName
```

További SQL utasítások

Bár a `select` a leggyakrabban használt SQL utasítás, léteznek más parancsok is a nyelvben. Íme a legfontosabbak:

- Az `Insert`-el új rekordokat adhatunk a táblához. Általában lehetőség van egy `select` eredménytábla beszúrására is, de ezt a Database Desktop nem támogatja.
- Az `Update` a táblában lévő adatok megváltoztatására szolgál.
- A `Delete`-tel sorokat törölhetünk ki a táblából.
- A `Create Table` parancs egy új táblát hoz létre.
- Az `Alter Table` utasítással egy már létező tábla szerkezetét módosíthatjuk, mezők törlésével vagy hozzáadásával.

Tárgymutató

A

ablakkezelő függvény I-443
About *lásd* **névjegy**
absztrakció II-479
ActiveForm I-5, I-28, II-211
  többlapos II-216
ActiveX II-113, II-175
  használata Delphiben II-177
  írása II-183
  kontrolok II-175
  TOleControl II-180
  tulajdonságlap II-190
  vezérlők telepítése II-178
adatbázis-belépés I-837
adatbázishibák I-839
adatcsere II-427
adatfolyamok II-417
adatmodulok I-819
adatszótár I-830
adattípusok I-104
Alignment paletta I-59
alkalmazás-ablak I-446, II-226
alkalmazás-példányok II-230
  másolat felkutatása II-231
állapotinformációk II-277
  inicializálós állományok II-277
  Registry II-281
állománytípusok I-135
AppendMenu *lásd* **rendszermenü**
Application objektum II-223
  események II-224
  tulajdonságok II-223
assertion II-308
Assigned I-257
attribútumhalmaz I-832

B

BDE I-728
  paraméterezett I-748
bejárási sorrend I-345
  Edit Tab Order I-345
Blaise Pascal I-101
BOF (Beginning of File) I-762
Borland C++ Builder I-18
Borland Database Engine *lásd* **BDE**
Borland Resource Workshop II-332

C

callback függvények I-178
Canvas *lásd* **vászon**
ciklusok I-158
  for I-158
  repeat I-159
  while I-159
class *lásd* **osztály**
class factory *lásd* **osztálygyárak**
clipboard *lásd* **vágólap**
CoCreateGuid II-122
Code Insight I-62
  Code Completion I-62
  Code Parameter I-62
  Code Templates I-62
  ToolTip Expression Evaluation I-62
COM II-113
  GUID II-122
  in-process szerver II-115
  interfészek II-126
  IUnknown II-120
  kliens II-131
  marshalling II-115
  out-of-process szerver II-115
  REGSVR32EXE II-131
  TComObject osztály II-126
  TComObjectFactory II-129
Component menü I-51
Component Object Model *lásd* **COM**
context-menu *lásd* **előugró menük**
CoolBar I-567, I-590
CreateComObject II-125
CreatePolygonalRgn II-34
CreateWindow I-456, II-18
CreateWindowEx I-456
critical section *lásd* **kritikus szakaszok**

D

Database Explorer I-730
Database Form varázsló I-750
Database menü I-51
dátum- és időtípusok I-110
  Date I-111
  DateTimePicker komponens I-114
  DateTimeToStr I-111
  DateTimeToString I-111
  DateToStr I-111
  DayOfWeek I-111
  DecodeDate I-111
  DecodeTime I-111
  EncodeDate I-111
  EncodeTime I-111

FormatDateTime I-111
Now I-111
StrToDate I-111
StrToDateTime I-111
StrToTime I-111
TDateTime I-110
Time I-111
TimeToStr I-111
DCOM II-157
DDE II-443
csatolások II-449
kapcsolat tervezési időben II-447
párbeszédek II-443
szerver és kliens II-443
debugger *lásd* **hibakereső**
Delphi-verziók I-37
Client/Server Suite I-37
Professional I-37
Standard I-37
destruktor I-190, II-32
dinamikusan szerkesztett könyvtárak *lásd* **DLL**
DLL II-83
a dinamikus szerkesztés II-83
erőforrások II-86, II-104
export II-89
hívása II-97, II-102
hívása futásidőben II-106
készítés C++-ban II-91
készítés Delphiben II-95
stdcall II-89, II-94
szerepe II-83
Dr. Niklaus Wirth I-101
drag and drop I-338, I-701
DTopics I-40

E

Edit menü I-42
egységek I-197, I-210
általános szerkezet I-198
elágazások I-156
case I-158
if I-156
eljárások I-165
eljárás-típusok I-175
elődeklaráció I-172
előugró menük I-434
módosítása I-436
EnumWindows II-234
Environment Options párbeszédablak I-52
EOF (End Of File) I-762
erőforrás szerkesztők II-330
erőforrások II-329
közvetlen beolvasása II-334
értékadás I-102
érvényesség I-202
globális rejtett változók I-202
globális változók I-202
lokális változók I-202
események I-281, II-33, II-240
eszköztár I-567, I-586
combo box I-580
mozgatás I-582
Exception osztály I-300
expert I-79
external *lásd* **külső deklarációk**

F

fájlműveletek II-401
szövegfájlok II-402
szövegkonverter II-406
típusos fájlok II-410
fastcall I-171
felhasználói adattípusok I-122
névtelen típus I-123
felsorolt típusok I-125
feltételes fordítás II-306
File menü I-40
file típus II-401
FindWindow II-231, II-234
fókusz I-341
form I-5
készítése I-5
Form Designer I-54
Format függvény I-145
formázott adatbevitel I-347-I-353
FormMouseDown I-333
formok I-443, I-601, I-664
aktiválása II-229
átlapolt (overlapped) I-445
automatikus készítés I-468
bezárása I-469
billentyűzet-események I-472
egéresemények I-475
felbukkanó (pop-up) I-445
felosztása I-664, I-686
gyermekablakok (child windows) I-445
kereflikonok I-454
keretstílus I-451
kliensterület I-465
maximalizálás I-463
minimalizálás I-463
rajzolás I-479
stílusok I-447, I-456
topmost form I-448
FreeLibrary II-105
függvények I-165

Result I-165
visszatérési érték I-165
futási idejű típus információ I-228
as operátor I-228
is operátor I-228

G

GetClipboardData II-439
GetDialogBaseUnits II-48
GetFileVersionInfo II-362
GetFileVersionInfoSize II-362
GetKeyState I-595
GetLastError II-105
GetParent I-446 *lásd* **tulajdonos**
GetProcAddress II-106
GetStartupInfo II-231
GetSystemMenu I-419, II-228
GetTextExtent II-48
GetWindowLong II-226
GlobalAlloc II-439
GlobalLock II-439
GlobalMemoryStatus II-324
görgetés I-655
grafikus listák I-524
grafikus menüpontok I-407
grid *lásd* **rácsok**
GUID II-122

H

halmaz típusok I-127
háttérszámítás II-250
Help *lásd* **Súgó**
hibakeresés II-291
~i információ II-292
ActiveForm-ban II-292
DLL-ben II-292
értékek vizsgálata II-299
terminálablak II-304
töréspontok II-293
VCL forráskódban II-303
hibakereső I-68
töréspont I-68
hint *lásd* **tippek**
HTML II-197
böngészés II-197
formátum II-202
készítés II-202, II-210
oldal generálása II-203
táblák készítése II-206

I, J

icon tray *lásd* **ikontartó**
időzítő II-242, II-253
ikonok II-336
ikontartó II-339
InterBase szerver I-802
interfész I-263
Internet II-197
InvalidateRect I-489, I-552
IUnknown II-120
IUnknown interfész I-265

jelentések II-384, II-390
jelölőnégyzet I-357

K

kétirányú eszköz I-22
kivételkezelés I-230, I-300
at I-232
except I-230
Exception osztály I-234
finally I-230, I-237
raise I-230
try I-230
verem I-236
kódparaméter I-64
kódsablonok I-63
kollekciók I-316
Komponens varázsló II-10
komponensek I-327, I-495
csomagok II-6, II-13
dinamikus létrehozása I-246
fájlkezelés II-414
használata I-8
komponens paletta I-57
komponenspaletta I-8
létrehozása II-5
osztály típusú tulajdonságok II-28
tömbtulajdonság II-44
komponens-sablonok I-29, I-51
komponens-szerkesztők II-67
regisztrálása II-71
konstansok I-103
konstruktor I-190
könyvjelzők I-60
kritikus szakaszok II-268, II-271, II-272
külső deklarációk I-174
kurzor II-346
készítése II-347
Screen komponens II-347
változtatása II-350

L

lekérdezés I-733, I-744, I-793, II-487
listapanel *lásd* **TListBox**
LoadAccelerators II-335
LoadBitmap I-510, II-335
LoadCursor II-335, II-349
LoadIcon II-335
LoadLibrary II-104
LoadMenu II-335
LoadResource II-335
LoadString II-335

M

master-detail I-791
MDI I-707
- gyermekablak I-708, I-715
- keret I-710
- szülőablak I-709
- Window menü I-713

megjegyzések I-147
memóriakezelés II-321
Menu Designer I-387
menük I-385
- grafikus menüpontok I-407
- gyorsbillentyűk I-388
- Menu Designer I-387
- módosítása futásidőben I-395
- owner-draw menüpontok I-415
- pipa használata I-396
- rádió menüpontok használata I-399

menüsablonok I-389
MessageBeep II-457
MessageDlg I-14
metaclass I-260
metódus I-172, I-185
metódus-mutatók I-253
mező I-751, I-753
- számított mező I-757

modális form I-605
ModifyMenu I-413, I-415
multimédia II-457
- CD használata II-470
- hangfájlok lejátszása II-465
- videóklipek lejátszása II-467

Multiple Document Interface *lásd* **MDI**
multitasking II-241
mutató típus I-133
- Dispose I-133
- New I-133
- nil I-133
- típus nélküli mutató I-135

mutex II-236, II-268, II-271

N

nem modális form I-602
nem vizuális komponens készítése II-51
névjegy I-633
nyitóképernyő *lásd* **splash screen**
nyomtatás II-367
- adatbázis II-381
- form kinyomtatása II-367
- nyomtatási kép II-372
- Printer objektum II-371
- saját párbeszédablak II-368
- szabványos párbeszédablakok II-369
- szöveg nyomtatása II-378

O

Object Browser I-69, II-310
Object Inspector I-6, I-58
- események I-6, I-12
- Object Selector I-58
- tulajdonságok I-6

Object Linking and Embedding *lásd* **OLE**
Object Pascal I-101
Object Repository I-79
- form sablonok hozzáadása I-94
- opciók I-95
- testreszabás I-92

Object Selector I-58
objektum I-183
objektum-hivatkozási modell I-186
OLE II-113
OLE Automation II-113, II-145
- kliens II-156
- szerver II-150
- szerver regisztrálása II-156

OLE dokumentum II-159
- beágyazás II-159
- csatolás II-159
- eszköztárak II-166
- fájlkezelés II-169
- menü II-164
- párbeszédablakok II-167

OOP I-183
operátorok I-153
öröklődés II-480
osztály I-183
- absztrakt metódusok I-224
- destruktor I-190
- dinamikus metódusok I-223
- információ-rejtés I-194
- inherited I-222
- késői kötés I-218
- kompatibilitás I-215

konstruktor I-187, I-190
metódus I-185
öröklés I-211
override I-221
példány I-187
polimorfizmus I-218
private I-195
protected I-195
public I-195, I-196
published I-196
virtuális metódusok I-222
osztálygyárak II-125
osztályhivatkozás I-248
osztálymetódusok I-248
osztályok II-479
osztályreferenciák I-259
owner *lásd* **tulajdonos**

P

Package Editor II-13
paraméterátadás
cím szerint I-166
konstans paraméterek I-167
nyitott tömb paraméterek I-167
paraméterezett lekérdezések I-748
parancssor II-238
párbeszédablak mértékegységek II-48
párbeszédablakok I-611
bezárása I-614
modális I-611
nem modális I-615
parent window *lásd* **szülőablak**
párhuzamos programozás II-256
PlaySound II-462
pointer *lásd* **mutató típus**
polimorf formok I-646
polimorfizmus I-218, II-481
PostMessage II-234, II-241, II-250
precedencia szabály I-153
Project menü I-49
projekt
állomány I-27
mentése I-8
Project Manager I-27, I-65
Project Option párbeszédablak I-8
property editor *lásd* **tulajdonság-szerkesztők**
PtInRegion II-34

Q, R

QuickReport komponensek II-384, II-396

rácsok I-542
rádiógomb I-357
csoportosítás I-358
raszterműveletek II-247
régió II-34
rejtett képernyő I-634
rekord I-787
hozzáadása I-787
rekord típus I-131
változó rész I-132
ReleaseCapture I-504
rendszermenü I-419, II-227
testreszabása II-227
ReportSmith II-388
makró II-393
resource string konstansok I-104
resourcestring II-354
részintervallum típus I-124
RTTI *lásd* **Futási idejű típus információ**
Run menü I-50

S

Screen objektum I-545, II-273
képernyőinformációk II-273
Search menü I-45
SelectNext I-257
Self kulcsszó I-245
Sender paraméter I-289
SendMessage II-241
SetActiveWindow II-232
SetCapture I-504
SetClipboardData II-439
SetCursor II-352
SetForegroundWindow II-232, II-343
SetMenuItemBitmaps I-408
SetWindowLong II-226
SetWindowPos I-449
SetWindowText I-116
shell interfész II-135
Shell_NotifyIcon II-339
Show Compile Progress I-18
sorszámozott típusok I-105
Dec I-108
High I-109
Inc I-108
Low I-109
Odd I-109
Ord I-109
Pred I-109
Succ I-109

splash screen I-637
SQL I-733, II-487
SQL kiszolgáló I-801
státuszsor I-567, I-593
stdcall II-236
string típus I-136
ANSIString I-137
átalakítások I-141
C-stílusú karaktertömbök I-140
formázás I-144
hivatkozásszámlálás I-139
hosszú stringek I-136
Pascal stringek I-136
PChar típus I-140
SetLength I-139
Súgó I-38
F1 billentyű I-38
Help menü I-39, I-53
szálak II-256, II-269
prioritása II-261
szinkronizálása II-263
szemafor II-268
szinkronizálás II-261, II-263, II-268
szintaktikai elemek kiemelése I-150
szöveges bemenet I-341
szubrutin *lásd* **eljárások, függvények**
szülőablak I-445

T

Tab order *lásd* **bejárási sorrend**
tábla I-730, I-733, I-741
dinamikus létrehozása I-779
létrehozása I-778, I-782
összekapcsolás I-814
szűrés I-825, I-827
tábla könyvjelző I-767
Tag tulajdonság I-309
TAnimate I-514
tárolt eljárás I-802
TBatchMove I-802, I-808
TBevel I-498
TBitBtn I-496
TBookmark I-767
TButton I-328
TCheckListBox I-375
TClipboard II-439
TCollection II-45
TCollectionItem II-45
TColorDialog I-337, I-624
TColorGrid I-553
TComboBox I-377
TComObject II-126
TComObjectFactory II-129
TComponent II-21
TComponentEditor II-68
TCriticalSection II-272
TCustomLabel II-42
TDataModule *lásd* **adatmodulok**
TDataSource I-732
TDateTimePicker I-562
TDBEdit I-742
TDBGrid I-738
TDBNavigator I-743
TDrawGrid I-542
TeeChart I-554
TField I-751
TFindDialog I-624
TFontDialog I-354, I-622
TGraphicControl II-21
THeaderControl I-691
This *lásd* **Self kulcsszó**
TIExpert II-72
TImage I-498, I-517
TImageList I-539
TInterfacedObject osztály I-265
tipizált konstansok I-103
tippek I-568, I-574, I-594, I-596
típuskonverzió I-117
Chr I-117
FloatToDecimal I-118
FloatToStr I-118
FloatToStrF I-118
FloatToText I-118
FloatToTextFmt I-118
Int I-117
IntToHex I-117
IntToStr I-117
Ord I-117
Round I-117
Str I-118
StrPas I-118
StrPCopy I-118
StrPLCopy I-118
StrToFloat I-118
StrToInt I-117
StrToIntDef I-118
TextToFloat I-118
Trunc I-117
Val I-118
típuskönyvtárak II-150
Type Library Editor II-152
típusmegfeleltetés (casting) I-116
TLabel I-334
TListBox I-363
TListView I-539
TMediaPlayer II-464
TMemo I-354
többszálúság II-256

többszörös öröklődés I-263
TObject osztály I-187, I-289
TOleContainer II-161
TOleControl II-180
tömb típusok I-129
dimenziók I-130
indexhatár I-129
Tools menü I-52
ToolsAPI II-61, II-75
TOpenDialog I-620
töréspontok II-293, II-301
Breakpoint List ablak II-296
feltételes II-302
TOutline I-528
TOutlineNode I-536
TPageControl I-665
TPaintBox I-491
TPanel I-694
TPrinter II-371
TQuery I-732
tranzakció I-731, I-834
TRegIniFile II-282
TRegistry II-282
TReplaceDialog I-625
TRichEdit I-355
TScrollBar I-379
TSession I-771
TShape I-491
TSpeedButton I-496
TSplitter I-686
TStoredProc I-732, I-734
TStringGrid I-542
TStringList II-45
TTabbedNotebook I-677
TTable I-732
TTabSet I-670
TTabSheet I-665
TThread II-257
TToolbar *lásd* **eszköztár**
TTrackBar I-380
TTreeView I-532
tulajdonos I-445
tulajdonságok I-273
OOP I-276
write I-274
tulajdonság-szerkesztők II-61
telepítése II-65
típusok II-61
Turbo Pascal I-101
TVarRec I-169
TWinControl II-21
TypInfo egység II-77

U

unit *lásd* **egységek**
üzenetablakok I-626
üzenetek
feldolgozása háttérben II-254
üzenetkezelők I-223

V

vágólap II-427
komponensek másolása II-435
speciális adat másolása II-439
valós típusok I-109
változó elnevezési szabályok I-10
változók I-101
var I-102
varázslók I-86
Application Wizard I-89
Database Form Wizard I-87
Dialog Wizard I-91
készítése II-71
varázsló DLL-ek I-96
variant típus I-118
TVarData I-120
vászon I-480
VCL I-289
általános események I-315
általános metódusok I-312
általános tulajdonságok I-302
forráskód I-320
hierarchia I-293
kontrolok I-295
VCL forráskód II-303
VerQueryValue II-363
verziószámok II-361
View menü I-48
Visual Basic for Applications II-102
Visual Query Builder I-811
vizuális form öröklés I-642

W

WaitForSingleObject II-236
WaveOutGetNumDevs II-459
Web Deploy I-30
WebModule II-210
window handle I-443
window procedure *lásd* **ablakkezelő függvény**
Windows handle-k I-114
Windows komponensek I-327
Windows rendszermemória II-323
  rendszererőforrások II-324
Windows üzenetkézbesítés II-241
WinSight II-241, II-312
with utasítás I-162
wm_Create II-18
wm_DrawItem I-415
wm_GetMinMaxInfo I-466
wm_HScroll I-659
wm_MeasureItem I-415
wm_NCCalcSize I-465
wm_NCHitTest I-465
wm_NCLButtonDown I-465
wm_NCPaint I-465

$DESIGNONLY II-6, II-67
$RUNONLY II-6
DFM I-24

Delphi 3 Mesteri Szinten a piacon elérhető Delphi könyvek közül a legjobb kritikát kapta. A könyv tartalmát teljes mértékben a Delphi 3-hoz igazították, így az olvasó egy minden tekintetben jól használható könyvet vehet a kezébe. A Delphi programozás egyik elismert mestere, Marco Cantù a legérthetőbb és legátfogóbb művet készítette el e témáról.

A Delphi 3 Mesteri Szinten igazi értéket ad mind a kezdő, mind a haladó programozók kezébe:
– A 32 bites Delphi programozás exkluzív bemutatása
– Több száz mintaprogram tanulmányozása, felhasználása
– A 32 bites kontrolok, a párhuzamos programozás, az OLE és a Registry használata és sok egyéb érdekes téma.

Minden OOP lehetőséget felfedezhet a Delphi 3-ban.
Megtanulhatja az osztályok, objektumok, unitok, metódusok és interfészek használatát az Object Pascal nyelv segítségével. A könyv megmutatja, hogyan lehet használni a vizuális form leszármaztatást, az Object Repository-t és további új, hatékony eszközöket.

Mesteri szintre vezeti az olvasót a Windows 95 és a kliens/szerver programozás művészetében.
Bemutatja az Interneten is használható ActiveX és ActiveForm komponenseket; bemutatja, hogyan használhatjuk a form stílusokat, kereteket, felbontás, méretezést és elhelyezést; bemutatja, hogyan készítsünk felhasználóközelibb programot grafikai elemekkel és a TChart komponenssel; bemutatja, hogyan készítsünk eszközsorokat, CoolBar-okat, státuszsorokat; és bevezet minket a menük használatának mélyebb szintjeire, mint például a tulajdonos által rajzolt menük készítése.

Hogyan készítsünk saját komponenst?
A könyv bemutatja, hogyan készítsünk komplex, az igényeinknek megfelelő komponenseket, és hogyan fejlesszünk tovább már létezőket. Részletezi az „packages" használatát, amivel könnyebben tudjuk használni és forgalmazni a komponenseket.

Adatbázis-kezelő alkalmazások készítése Windows 95 alá
Megismertet minket a Varázslók, adatmodulok, multirecord objektumok, lock-up fájlok, a DBGrid, és a QuickReport komponensek használatával, hogy még egyszerűbben és még hatékonyabban készíthessünk adatbázis-kezelő alkalmazásokat.

A CD melléklet tartalmazza a könyvben előforduló példaprogramokat, valamint azok teljes forrását, angol nyelvű Delphi magazinok számait (kóddal együtt), a szerző által készített segédprogramokat, melyek hasznosak lehetnek munkánkban, valamint felhasználható Delphi komponenseket (ingyenesek vagy demo/shareware változatok).

Felhasználói szint:
Kezdőtől profiig
Könyv típus:
Kézikönyv/referencia
Kategória:
Windows programozás

Eladási ára: 2860 Forint
CD melléklettel
ISBN 963 03 5349 0
9 789630 353496

SYBEX®

Kiskapu Virtuális Könyvesbolt
http:/www.kiskapu.hu/kiskapu/

KISKAPU
KISKAPU KIADÓ